

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



中国Solr领域的资深专家和布道师撰写，权威性毋庸置疑。

以实战为导向，全面、系统、细致、深入地讲解Solr的基础知识、核心技术、进阶知识和扩展知识。



兰小伟 著

*The Definitive Guide of Solr*

# Solr权威指南

## 上卷



机械工业出版社  
China Machine Press

## 内 容 简 介

本书作者是国内最早接触Solr的技术专家之一，多年一直在Solr的研究、实践和布道的路上不遗余力、乐此不疲。本书立足全球视野，综合Solr技术的最新发展和应用、从业人员的学习曲线，以及中英文资料的供给情况，给自己设定了一个极高的目标：力争在内容的全面性、系统性、深浅度和实战性上超越所有的同类书。从完成的结果上来看，我们的目标接近完成，Solr的基础知识、核心技术、进阶知识和扩展知识悉数包括在内。

全书一共16章，分为上下两卷：

上卷（第1~10章）

全面、系统地讲解了Solr的基础知识和核心技术。包括部署、配置、Solr Core、Solr DIH、全量导入、增量导入、索引、中文分词、查询组件、Solr Facet、高亮、查询建议，以及企业如何在真实的项目中使用Solr。不仅讲解了基本概念和使用方法，而且还分析了各组件的内部工作机制。

下卷（第11~16章）

细致、深入地讲解了Solr的高级知识和拓展知识。

高级知识部分包括：Solr的高级查询及其各种查询技巧，如函数查询、地理空间查询、Facet嵌套等；SolrJ、SolrCloud、Spring Data Solr的使用详解和工作原理；Solr的多种性能优化技巧，如索引的性能优化、缓存的性能优化、查询的性能优化、JVM和Web容器的优化，以及操作系统级别的优化。

拓展知识中首先讲解了Solr的一些比较生僻的知识点，如伪域、多语种索引支持、安全认证，以及Solr 6.x中的SQL接口和Streaming表达式等；然后讲解了Solr与MapReduce、HDFS、Hbase、Kafka、Flume、Storm、Spark等大数据技术的结合使用的集成方法。

實戰



*The Definitive Guide of Solr*

# Solr权威指南

上卷

兰小伟 著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Solr 权威指南 上卷 / 兰小伟著. —北京: 机械工业出版社, 2017.10  
(实战)

ISBN 978-7-111-58172-7

I. S… II. 兰… III. 搜索引擎—程序设计—指南 IV. TP391.3-62

中国版本图书馆 CIP 数据核字 (2017) 第 261602 号

## Solr 权威指南 上卷

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 何欣阳

责任校对: 殷虹

印 刷: 北京诚信伟业印刷有限公司

版 次: 2018 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 27.5

书 号: ISBN 978-7-111-58172-7

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

## Preface 序 言

Apache Solr 是使用最广泛的全文检索解决方案，大部分网站都在使用 Solr 来实现搜索功能。然而国内关于 Solr 的资料太少，无奈我只能一点点地啃 Solr 官方提供的 User Guide PDF 文档、Solr Wiki 以及一些纯英文的技术书籍，希望能够借由本书将我学习积累的所有经验倾情传授给那些由于学习 Solr 曲线太陡峭而束手无策的同学们。本书致力于帮助 Java 开发人员更简单、深入地学习 Solr。同时本书还提供了随书源码，其中包含大量可运行的示例代码。本书与随书源码搭配在一起学习会事半功倍！由于目前大数据、云计算的发展如火如荼，各种大数据生态框架如雨后春笋般涌现，给人一种无形的压力。为此，本书也介绍了 Solr 与大数据框架的集成，如果你正好有这方面的需求，希望本书能够给你带来帮助。

### 为什么写这本书

转眼间，我已经跌跌撞撞走过了 5 个年头，由起初的那个 Java 迷途小书童变身为程序员届的一根老油条，不由感慨万千。由于深谙一个非高校毕业的“正规军”一路走来有多么的艰辛，因此我一直秉持爱开源、爱分享的个性。这么多年来帮助过的程序员太多太多，本着一颗乐于助人的心，我不想大家重走我的弯路。从 2015 年 3 月中旬开始，我在 ITEye 技术社区发布与 Lucene 和 Solr 相关的技术博客，深受大家喜爱。每天联系、咨询我问题的网友越来越多。疲于应付的我，开始意识到仅靠一个人这样一对一地指导是行不通的。而且刚好 Solr 这方面的中文技术书籍在中国还是一片空白，于是萌生了写一本 Solr 中文书籍的想法，希望能够帮助更多的 Solr 技术爱好者。

2015 年 8 月我联系到了华章的杨福川，向他提出了写这本书的想法，得到了他的大力支持。我深知自己过往没有显赫耀眼的工作经历，在一些前辈面前还只是一个晚辈。因此，在创作本书的过程中，查阅了 Solr 官网提供的 Apache-Solr-Ref-Guide、Solr Wiki，并通读了《Solr in action》《Apache Solr 4 Cookbook》《Apache Solr Essentials》《Apache Solr High



Performance》等英文技术书籍。为了能够编写 Solr 与大数据集成相关章节，我又耗费了大量时间通读了《Apache Flume Distributed Log Collection for Hadoop》《Hadoop in Action》《HBase in Action》《Learning Spark》等大数据相关的英文技术书籍。写作本书的过程也成为本人学习提升的过程，为此我花费了整整 1 年的时间。资历尚浅仍可以通过自身努力来弥补，所以我时时刻刻以严谨缜密的态度对待写进书里的每一段文字，除了怀揣着对技术的一种敬畏之情，我知道我还必须为读者负责。

然而造化弄人，在 2016 年的 2 月份，我的颈部莫名其妙长了一个肿瘤，这严重影响了我身心健康。由于辗转于北京协和医院、解放军总医院等地投医救治，所以这本书的编写工作不得不临时中断。还好我没有放弃，于是在修养了半年之后，又进入了“挑灯夜读”的状态，开始夜以继日地赶稿子。因为已经立下了写书的豪言壮志，所以再苦再累我也是要写完的！由于生病，当初所在的公司要求我立即停薪修养，在看尽了世态炎凉之后，我毅然选择了辞职，打算专职将这本书写好，给读者一个交代。没有了经济来源，只靠自己多年来的积蓄维持生活。我顶着巨大的压力，在大病初愈的情况下，决定倾注全部精力打造这本书。很庆幸我坚持下来了。每天叫醒我的不是闹钟不是鸡汤，也不是其他竞争对手，而是我的决心，因为父母已两鬓白发，快要三十的我还孑然一身。所以我不能虚度光阴，需要为了我爱的人和爱我的人努力奋斗，从而改善他们的生活。这本书也算是给自己 30 岁生日提前备下的一份礼物，并借以纪念不悔的青春岁月。我知道和我有着类似经历的同学太多太多，因此希望这本书能够为学习 Solr 的你们带来帮助和鼓励：定好一个 Target，就永远不要放弃！

## 准备工作

随书提供了大量的示例代码（本书随书示例源码下载地址：<https://github.com/yida-lxw/solr-book>），其中涉及 MongoDB、ZooKeeper、Hadoop、HBase、Flume、Kafka、Storm、Spark、Scala 等知识点，不仅限于 Solr，所以对于 Java 初学者而言会有一定压力。尽管书中提供了部分大数据框架的集群搭建步骤，但是由于篇幅的限制不可能面面俱到，你还是需要另外查阅其他相关书籍或资料来补充大数据这方面的知识。由于随书源码是基于 Maven 构建的，因此你还需要掌握 Maven 的基本使用方法。为了尽最大努力满足大部分用户的需求，所以从第 14 章开始我将以 Solr 6.2.1 版本为例进行讲解，而 Solr6.x 是要求 JDK 1.8+ 版本的，那么在学习本书之前，你需要提前安装好 JDK 1.7 和 JDK 1.8。如果你有将 Solr 部署在 Tomcat 下的需求，那么你还应安装 Tomcat 环境。对于企业而言，SolrCloud 集群通常会部署在 Linux 环境下，因此本书 SolrCloud 部分是以 CentOS 6.5 为例进行讲解的，或许你还需要掌握 Linux 操作系统的基础知识以及一些 Linux 的常用命令。另外，由于 Solr 是基于 Lucene 构建的，因此

你最好拥有一定的 Lucene 基础再来学习本书内容会感觉更轻松。因为本书自始至终是以由浅入深的原则进行编写的, 尽量细致入微地讲解每一步。当然, Solr 源码是使用 Java 编写的, 这也要求你能够熟练掌握 Java 编程语言的知识, 并拥有良好的编码基本功以及编程悟性。而 Solr 中的数据往往来自于关系型数据库, 因此你最好是对关系型数据库有一定的了解。

## 如何阅读本书

全书分为上下两卷, 总共 16 章, 涵盖了 Solr 各个方面的知识点。本书从前到后按内容的难易程度以循序渐进的方式呈现出来。因此你只需要拥有足够的毅力将它阅读完, 当然最好是能够边读边上机实践, 就可以掌握 Solr。此外每章之间都是相互独立的, 如果你对于某章的内容已经非常熟悉, 那么可以直接跳过选择感兴趣的章节进行学习。当然还是建议大家能够通读本书, 系统的学习 Solr, 这样才会对 Solr 有一个更完整的理解, 为你日后从事 Solr 相关的开发工作打下夯实的基础。本书每章开头部分都列举了该章的主要知识点, 可以让你快速了解本章能够学习到的内容。虽然本书中演示的示例代码在随书源码中都可以找到, 但是我还是建议大家能够实际动手去敲一遍, 毕竟只有亲身实践过, 才能将遇到的各种问题真正悟透并彻底解决。这个过程虽是艰辛的, 但也是深刻的, 因为解决问题对于程序员来说就是积累经验的机会。

## 面向的读者

- Java 开发工程师;
- 架构师;
- Solr 技术爱好者;
- 各大高校或 IT 培训机构的学弟学妹们。

## 勘误与反馈

在编写本书的过程中, 尽管我倾注了大量时间与精力, 但是由于水平有限, 书中难免会存在不足与疏漏之处, 还请大家多多批评指正。如果你在阅读本书过程中有任何疑问或者建议需要向我反馈, 可直接发送 E-mail 至 736031305@qq.com 或者添加个人微信 (13476669029) 联系我。

## 致谢

不知道你拿到这本书的时间是哪一年哪一个季节, 但是对我来说, 这都是我在自己 30

岁之前完成的一个最大的心愿。这是国内真真正正全面介绍 Solr 技术的第一本中文书籍，很开心我做到了。

想感谢的人很多，首先要谢谢爸妈，在我生病期间无微不至地照顾我，并无条件地支持我。

谢谢一路以来理解并鼓励我的朋友和粉丝们，是你们让我不断坚持前行。

谢谢机械工业出版社华章公司的杨福川、高婧雅、李艺，在这一年当中对我写作的信任与帮助，没有你们辛勤的付出，就不可能有这本书的面市。非常开心和幸运能够与你们共同完成这样一本书籍。

谢谢我的 Java 启蒙老师习晨龙，是您带我进入了 Java 世界，从此我在汲取知识的路上甘之如饴。

谢谢在这么多年的工作中所有帮助过我的同事，我会一直记得你们。

最后需要感谢的还是我自己，感谢曾经的年少轻狂，感谢一直都存在的梦想，对于梦想我从来没有也永远不会放弃。所以如果你还有梦想，为了你爱的人，为了你自己，请永远不要放弃！



## Contents 目 录

### 序言

## 第1章 初识 Solr ..... 1

- 1.1 Solr 是什么 ..... 1
- 1.2 Solr 的历史 ..... 2
- 1.3 为什么要选择 Solr ..... 2
- 1.4 Solr 功能预览 ..... 3
- 1.5 Solr 下载 ..... 3
- 1.6 Solr 学习资源 ..... 5
- 1.7 Windows 平台下部署 Solr ..... 7
  - 1.7.1 部署 Solr 至 Jetty ..... 7
  - 1.7.2 部署 Solr 至 Tomcat ..... 13
- 1.8 Linux 平台下部署 Solr ..... 16
- 1.9 玩转 post.jar ..... 20
- 1.10 在 Eclipse 中编译 Solr 源码 ..... 25
- 1.11 本章总结 ..... 27

## 第2章 Solr 基础 ..... 28

- 2.1 Solr Core ..... 28
  - 2.1.1 Solr Core 简介 ..... 28
  - 2.1.2 Core 的基本管理 ..... 30

2.1.3 Core Http 接口 ..... 35

2.1.4 添加索引至 Core ..... 36

## 2.2 Solr DIH ..... 38

2.2.1 索引文件夹下的文本文件 ..... 38

2.2.2 索引 JSON/XML/CSV 文件 ..... 42

2.2.3 使用 Tika 索引 Word/Excel/  
PDF ..... 45

2.2.4 索引网络上的远程文件 ..... 52

2.2.5 索引 XML 文件 ..... 55

2.2.6 从数据库中导入数据至 Solr ..... 57

2.2.7 Solr DIH 总结 ..... 62

## 2.3 Solr Full Import 全量导入 ..... 78

## 2.4 Solr Delta-import 增量导入 ..... 80

## 2.5 Solr 索引 ..... 85

2.5.1 Lucene 索引原理 ..... 85

2.5.2 Lucene 中常见术语详解 ..... 87

2.5.3 创建 Solr 索引 ..... 98

2.5.4 Solr Cell ..... 99

2.5.5 Solr 索引去重检测 ..... 102

2.5.6 Solr 更新请求处理链 ..... 104

2.5.7 Solr 原子更新 ..... 105

2.5.8 使用 Luke 查看索引 .....	107	4.3.1 IK 分词器 .....	217
2.6 本章总结 .....	109	4.3.2 Ansj 分词器 .....	223
<b>第 3 章 Solr 配置</b> .....	110	4.3.3 MMSeg4J 分词器 .....	233
3.1 solr.xml 配置详解 .....	110	4.3.4 Paoding 分词器 .....	240
3.2 solrconfig.xml 配置详解 .....	112	4.3.5 Jcseg 分词器 .....	245
3.3 schema.xml 配置详解 .....	139	4.3.6 Ictclas 分词器 .....	258
3.3.1 Solr Schema 设计思想 .....	139	4.3.7 FudanNLP .....	259
3.3.2 Solr 眼里的世界 .....	139	4.3.8 HanLP .....	262
3.3.3 域分词 .....	140	4.3.9 Jieba 分词器 .....	266
3.3.4 Solr 的 schema 文件 .....	140	4.3.10 分词器使用建议 .....	268
3.3.5 Solr 的域类型 .....	141	4.4 本章总结 .....	270
3.3.6 Solr 的域 .....	153	<b>第 5 章 Solr 查询</b> .....	271
3.3.7 Schema API .....	157	5.1 Solr 查询概述 .....	271
3.3.8 Schemaless Mode .....	165	5.2 Solr 查询相关度简述 .....	273
3.4 data-config.xml 配置详解 .....	167	5.3 Solr 的查询语法解析器 .....	275
3.5 zoo.cfg 配置详解 .....	169	5.4 Lucene 的基本查询语法 .....	283
3.6 本章总结 .....	169	5.5 Solr 的标准查询语法解析器 .....	287
<b>第 4 章 Solr 分词</b> .....	170	5.6 Solr DisMax .....	288
4.1 分词的基本概念 .....	170	5.7 Solr eDisMax .....	291
4.1.1 理解 Analyzer .....	170	5.8 Solr 的其他查询语法解析器 .....	298
4.1.2 理解 Tokenizer .....	171	5.9 Query VS Filter Query .....	305
4.1.3 理解 TokenFilter .....	172	5.9.1 fq VS q .....	306
4.2 Solr 分词器 .....	172	5.9.2 Filter Query 缓存 .....	307
4.2.1 Analyzer .....	173	5.9.3 Filter Query 执行顺序 .....	308
4.2.2 Tokenizer .....	174	5.9.4 Post Filter .....	308
4.2.3 TokenFilter .....	182	5.10 Solr 返回结果 .....	309
4.2.4 CharFilter .....	202	5.10.1 设置响应输出格式 .....	309
4.2.5 Solr 自定义分词 .....	206	5.10.2 选择返回域 .....	310
4.3 中文分词器 .....	217	5.10.3 分页查询 .....	312
		5.11 Solr 排序 .....	313

5.11.1 根据域进行排序 .....	313
5.11.2 缺失值处理 .....	314
5.11.3 排序的内存占用 .....	315
5.12 调试查询结果 .....	315
5.12.1 返回调试信息 .....	315
5.12.2 开启调试模式 .....	316
5.13 本章总结 .....	316

## 第6章 Solr Facet .....

6.1 理解 Facet .....	317
6.2 Facet 简单示例 .....	319
6.3 Query Facet .....	326
6.4 Range Facet .....	328
6.5 FacetFilter .....	330
6.6 Multiselect Faceting .....	335
6.6.1 key .....	335
6.6.2 tag .....	336
6.7 本章总结 .....	339

## 第7章 Solr 高亮 .....

7.1 什么是 Solr 高亮 .....	340
7.2 Solr 高亮的工作原理 .....	342
7.2.1 Fragmenter .....	348
7.2.2 Scorer .....	349
7.2.3 Encoder & Formatter .....	349
7.3 Facet & Highlighting .....	350
7.4 高亮多值域 .....	351
7.5 高亮参数 .....	352
7.6 FastVectorHighlighter .....	355
7.7 PostingsHighlighter .....	356
7.8 本章总结 .....	358

## 第8章 Solr Query Suggestion 查询建议 .....

8.1 Spell-Check .....	361
8.1.1 Spell-Check 简单示例 .....	361
8.1.2 Spell-Check 查询组件 .....	362
8.2 Autosuggest .....	366
8.3 基于 N-Gram 实现 Autosuggest .....	369
8.4 基于用户行为实现 Autosuggest .....	371
8.5 本章总结 .....	375

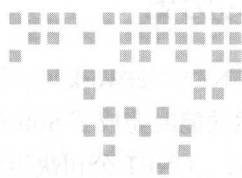
## 第9章 Solr Group 分组 .....

9.1 Result grouping VS Field collapsing .....	377
9.2 按照指定域分组 .....	377
9.3 每个分组返回多个文档 .....	381
9.4 按照 Function 动态计算值分组 .....	382
9.5 按照任意 Query 分组 .....	383
9.6 Group 的分页与排序 .....	383
9.7 Group & Facet .....	384
9.8 Group 分布式查询 .....	387
9.9 Group 缓存 .....	388
9.10 使用 Collapsing Query Parser 实现高效的 Field Collapsing .....	388
9.11 Solr Group VS SQL Group by .....	389
9.12 本章总结 .....	390

## 第10章 Solr 企业级应用 .....

10.1 Solr 源码编译与补丁应用 .....	391
10.2 部署 Solr .....	396
10.2.1 构建你自己的 Solr 发布版本 .....	397

10.2.2 Embedded Solr .....	397	10.8 如何与 Solr 交互 .....	414
10.3 Solr 硬件要求与系统配置 .....	397	10.8.1 使用 REST API 与 Solr 交互 .....	415
10.3.1 内存和 SSD .....	397	10.8.2 使用 SolrJ 与 Solr 进行 交互 .....	415
10.3.2 JVM 配置 .....	398	10.9 监控你的 Solr .....	418
10.3.3 思考 Solr 索引与查询性能 .....	401	10.9.1 Solr 的性能统计 .....	418
10.4 Solr 数据批量导入 .....	405	10.9.2 Solr 的缓存性能 .....	419
10.5 Solr Shard 与 Replication .....	406	10.9.3 Solr JMX .....	419
10.5.1 Shard .....	406	10.9.4 Solr 日志 .....	424
10.5.2 Replicate .....	408	10.9.5 Solr 负载测试 .....	424
10.6 Core 管理 .....	410	10.10 Solr 版本升级 .....	428
10.7 Solr 集群管理 .....	412	10.11 本章总结 .....	428
10.7.1 Solr Ping 健康检测 .....	412		
10.7.2 Solr 配置文件管理 .....	413		



## 初识 Solr

通过第1章，你将可以学习到如下内容：

- ❑ Solr 是什么；
- ❑ Solr 的历史；
- ❑ 为什么要选择 Solr；
- ❑ Solr 下载安装部署；
- ❑ Solr 源码编译。

### 1.1 Solr 是什么

Solr 是目前非常受欢迎的基于 Apache 开源组织下 Lucene 开发的一个开源高性能的企业级搜索平台。Solr 具有高度可靠性、可扩展性、可容错性的特点，提供了分布式索引、索引备份、查询负载均衡、自动故障转移和恢复，以及集中配置等功能。Solr 使 Web 搜索大放异彩，许多知名企业都在使用它，如大名鼎鼎的 eBay、Instagram 等。

如今我们已经步入信息大爆炸时代，信息量如此庞大，渴望从信息海洋中快速检索相关性内容的诉求已经随着以 Google 为代表搜索引擎的崛起而变得越来越强烈，使用 Solr 完全可以帮助企业满足用户的搜索需求，这也是 Solr 的设计初衷。Solr 是完全开源的，且易于安装部署，还提供了一个基于 AngularJS 开发的 Web 管理后台界面，以便用户对 Solr 进行可视化管理。此外 Solr 拥有一群庞大而活跃的开发贡献者和支持者，如果你在学习过程中有任何问题，都可以轻松得到帮助。Solr 如今正以惊人的速度不断迭代更新，截至本书执笔之前，Solr 已经更新到 5.3.1，这也从侧面印证了 Solr 如今的火热程度。

## 1.2 Solr 的历史

2004 年, CNET NetWorks 公司 (www.cnetnetworks.com) 的 Yonik Seeley 工程师为公司网站开发搜索功能时完成了 Solr 的雏形。起初 Solr 只是 CNET 公司的内部项目。

2006 年 1 月, CNET 公司决定将 Solr 源码捐赠给 Apache 软件基金会, 希望 Apache 能帮助 Solr 解决一些关于组织、法律和金融等方面的问题, 这样 Solr 就像 Apache 旗下其他新项目一样进入了孵化器。2007 年 1 月, Solr 顺利孵化成为 Apache 旗下一个独立的顶级项目。通过 1 年时间的不断累积和稳步发展, Solr 吸引了一大批用户群, 其中不乏代码贡献者、参与者。即便是作为一个全新的开源项目, Solr 依然被应用于一些高流量的网站中。

2008 年 9 月, Solr 1.3 发布了新功能, 其中包括分布式搜索和性能增强等功能。

2009 年 1 月, Yonik Seeley、Grant Ingersoll 和 Erik Hatcher 一起联手成立了 Lucidworks 公司。Lucidworks 成为了第一家为 Solr 提供商业和技术支持的公司。自此, Solr 提供的服务开始全面丰富起来。

2009 年 11 月, Solr 1.4 版本发布, 此版本对索引、搜索、Facet 等方面做了改进, 比如提高了对 PDF、HTML 等富文本文件的处理能力, 还推出了许多额外的插件。

2010 年 3 月, Lucene 和 Solr 项目合并, 自此, Solr 成为了 Lucene 的子项目。产品现在由双方的参与者共同开发。

2011 年, Solr 改变了版本编号方案, 以便与 Lucene 匹配。为了使 Solr 和 Lucene 有相同的版本号, Solr 1.4 的下一版本号变为 3.1。

2012 年 10 月, Solr 4.0 版本发布, 新功能 SolrCloud 也随之发布。

## 1.3 为什么要选择 Solr

Solr 的目标是打造一款企业级的搜索引擎系统, 也更接近于大众所熟知的搜索引擎系统, 作为一种搜索引擎服务, 它可以通过各种 API 在你的应用上使用, 而不需要将搜索逻辑耦合其中。Solr 可以根据配置文件定义数据解析的方式, 更像是一个搜索框架, 同时也支持主从、Core 热切换等操作, 还添加了对 Highlight、Group、Facet 等搜索引擎常见功能的支持。

自 Solr4.0 版本开始, Solr 引入了具有里程碑意义的 SolrCloud 分布式索引解决方案, 其基于 ZooKeeper 的分布式搜索方案, 主要思想是使用 ZooKeeper 作为集群的配置信息中心, 保证了 Solr 在可伸缩性与可容错性方面有很好的表现。当你的索引数据越来越庞大, 单机无法承受时, 使用 Solr 的 SolrCloud 部署方式也不失为一种选择。

对于开发人员而言, Solr 提供的各种功能亮点足以震撼到你, 而且它是开源的, 意味着你可以很轻易地得到它的源码并随意修改以满足自己的特殊需求, 轻松清除了你学习过程中的绊脚石。

作为 Solr 的系统管理员, Solr 使用 AngularJS 开发了一个美观易用的 Web 后台界面。

你可以在这里对 Solr 进行各种管理配置，如配置检查、查询测试、Solr 监控。

作为技术经理或者 CEO，你可能不关心技术细节，而只关心如何使用 Solr 这门技术，包括需要投入多少人力物力的成本，以及使用风险评估等。首先，Solr 是开源免费的，其次，Solr 的开源社区异常活跃，而且 Solr 是由 Apache 软件基金会组织进行开发和管理的，不用担心 Solr 哪天突然没人更新维护了，更不用担心遇到问题无从寻求帮助。

## 1.4 Solr 功能预览

让搜索解决方案具有良好的用户体验是 Solr 使用者的最大诉求，现在让我们快速浏览下 Solr 为了实现搜索功能的良好用户体验都提供了哪些功能：

- ☐ 灵活的查询语法；
- ☐ 支持各种格式文件（Word，PDF）导入并索引；
- ☐ 支持数据库数据导入并索引；
- ☐ 分页查询和排序；
- ☐ Facet 维度查询；
- ☐ 自动完成功能；
- ☐ 拼写检查；
- ☐ 搜索关键字高亮显示；
- ☐ Geo 地理位置查询；
- ☐ Group 分组查询；
- ☐ SolrCloud。

至此，我想你应该已经清楚：Solr 是用来做什么的，Solr 支持对什么类型的数据进行索引，Solr 不是数据库的替代品。那就赶紧随我立即下载 Solr 安装包，并将它运行于你的本地机器上，身临其境地感受下 Solr 的魅力吧！

## 1.5 Solr 下载

因为 Solr 是基于 Java 语言开发的，且 Solr 5.x 要求 JDK 的最低版本必须是 JDK 7，所以在开始 Solr 学习之旅之前，建议首先在你的本机安装好 JDK 7 或者 JDK 8。如果你是 Java 初学者，从未在本机搭建过 Java 开发环境，建议先买一本 Java 基础入门级书籍进行学习，而后再回到 Solr。至于 JDK 如何安装以及 JDK 环境变量如何配置，具体可从网上查询相关内容，这里不再赘述。

在我写本书时，Solr 的最新版本是 5.3.1，所以本书主要以此版本为例进行讲解，有可能当你看到本书时，Solr 已经更新，如果跟本书使用的版本号不一致，请不要有顾虑，其实使用方式基本差不多。接下来是 Solr 的下载步骤：打开浏览器访问 Solr 官网（<http://lucene.apache>。

org/solr/), 首先会看到顶部一排红色背景的导航栏, 单击其中的 DOWNLOAD 超链接进入 Solr 的下载引导页面, 此页面会有一个 3 秒的停顿, 然后会自动跳转到下载页面, 如图 1-1 所示。



图 1-1 Solr 下载演示图

如果希望下载 Solr 当前最新版本, 你可以等待 3 秒让页面自动跳转, 或者直接单击 Latest version 段落部分的超链接直接进入 Solr 当前最新版本下载页面, 如图 1-2 所示。



图 1-2 Solr 下载地址演示

如果你不希望下载当前的最新版本, 而需要 Solr 4.x、Solr 3.x 甚至 Solr 1.x 的版本, 可以单击 Past versions 段落部分的超链接进入 Solr 历史版本下载界面, 然后挑选你想要下载的版本进行下载。对于 Java 新手, 看到如图 1-3 的下载界面可能会有些迷茫, 不知该下载哪一个文件, 所以这里我稍微做下说明。

其中文件名包含 src 字符的文件表明这是 Solr 的源码压缩包, 直接以 solr-version.zip 或 solr-version.tgz 命名的文件是 Solr 的安装包, zip 对应 Windows 平台, tgz 对应 Linux 平台, 而以 asc/md5/sha1 结尾的文件都是校验文件。为了确保下载的文件是安全的, 没有被植入病毒或木马, 所以官方提供了一致性校验文件。文件的一致性校验, 基本原理就是通过加密算法对



文件生成一系列对应的唯一值，然后同文件发布者提供的值进行比较，从而判断两者是否一致。ASC、MD5 和 SHA1 就是目前使用最为广泛的加密算法。如果你非常关心下载的文件是否安全，那么可以下载相应的校验文件。Windows 平台下有几款文件一致性校验的工具，如：HashCalc、WinMD5、Hasher。为了方便日后学习，建议大家将 Solr 的源码包也一并下载下来。

archive.apache.org

This site contains the historical archive of old software releases.

For current releases, please visit the [mirrors](#).

Name	Last modified	Size	Description
<a href="#">Parent Directory</a>		-	
<a href="#">changes/</a>	2015-09-23 11:31	-	
<a href="#">KEYS</a>	2015-09-16 21:04	142K	
<a href="#">solr-5.3.1-src.tgz</a>	2015-09-16 21:04	37M	
<a href="#">solr-5.3.1-src.tgz.asc</a>	2015-09-16 21:04	842	
<a href="#">solr-5.3.1-src.tgz.md5</a>	2015-09-16 21:04	53	
<a href="#">solr-5.3.1-src.tgz.shal</a>	2015-09-16 21:04	61	
<a href="#">solr-5.3.1.tgz</a>	2015-09-16 21:04	129M	
<a href="#">solr-5.3.1.tgz.asc</a>	2015-09-16 21:04	842	
<a href="#">solr-5.3.1.tgz.md5</a>	2015-09-16 21:04	49	
<a href="#">solr-5.3.1.tgz.shal</a>	2015-09-16 21:04	57	
<a href="#">solr-5.3.1.zip</a>	2015-09-16 21:04	136M	
<a href="#">solr-5.3.1.zip.asc</a>	2015-09-16 21:04	842	
<a href="#">solr-5.3.1.zip.md5</a>	2015-09-16 21:04	49	
<a href="#">solr-5.3.1.zip.shal</a>	2015-09-16 21:04	57	

图 1-3 Solr 下载目标文件

## 1.6 Solr 学习资源

为了方便大家学习，Solr 官方提供了很多学习资源，如图 1-4 所示。

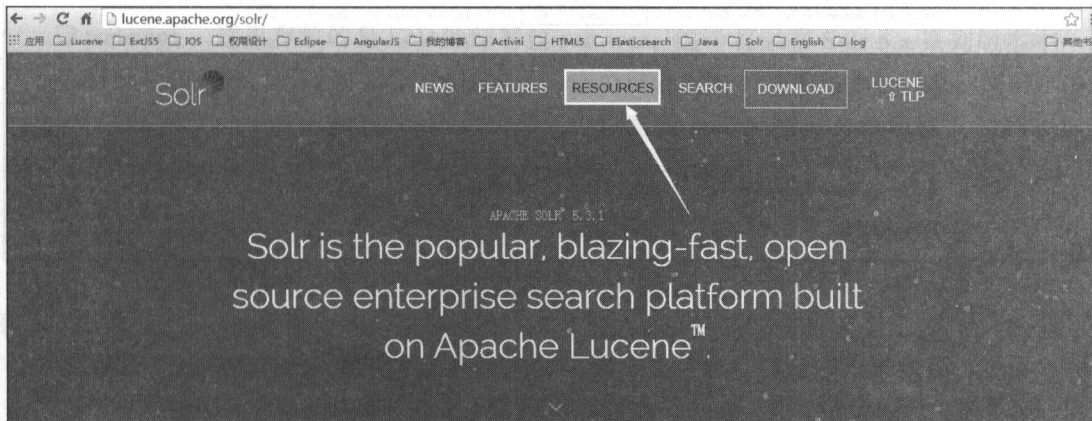


图 1-4 Solr Resources

打开 Solr 学习资源页面，你将看到官方大致提供了 4 个资料：

- 1) Solr Quick Start (即 Solr 快速上手教程)；
- 2) Solr 官方使用指南，获取方式如图 1-5 所示；

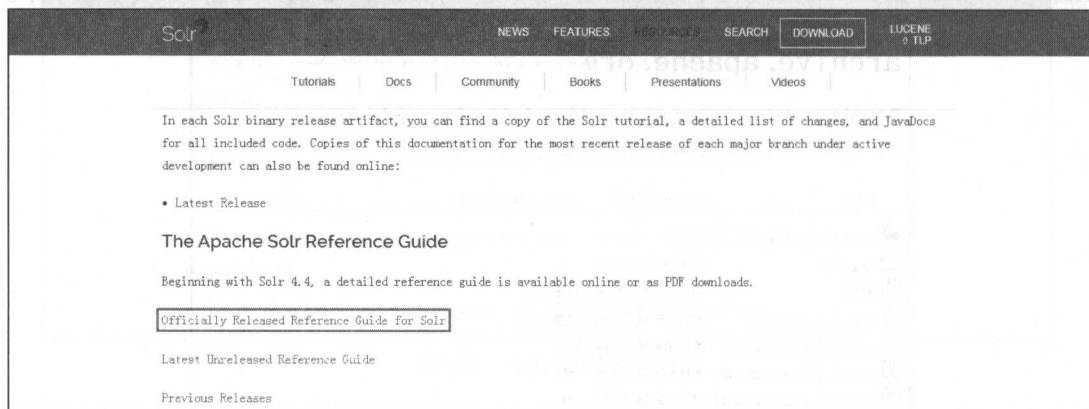


图 1-5 Solr 官方使用指南

- 3) Solr 官方 Wiki，如图 1-6 所示；

4) Solr 相关的英文书籍，如《Solr in action》《Solr Cookbook》，这些英文书籍都可以轻易地通过搜索引擎获取。

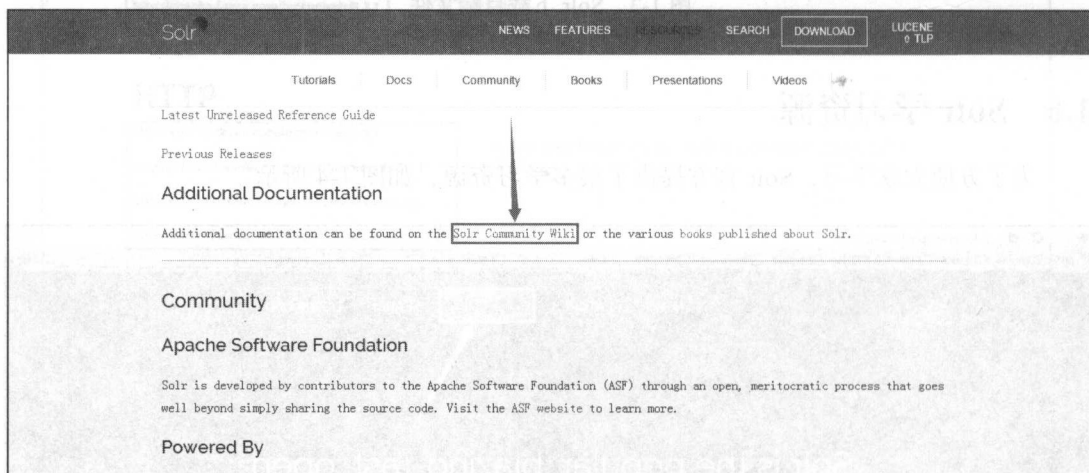


图 1-6 Solr 官方 Wiki

不过，官方提供的资料全是英文资料，这对于英文不太给力的同学来说，理解起来会有一定难度，所以会希望有一本中文教程能用图文并茂的方式以及通俗易懂的语言把这门晦涩难懂的热门技术讲解透彻。这也正是我写这本书的初衷，希望对大家学习 Solr 带来帮助。

## 1.7 Windows 平台下部署 Solr

Windows 是我们平时学习和开发时最常用的平台，所以首先我将带领大家学习如何在 Windows 平台下部署 Solr，从而更直观地感受 Solr 的魅力，这也是 Solr 初学者最迫切的需求。

### 1.7.1 部署 Solr 至 Jetty

从 Solr 官网下载完 Solr 安装包后，你将得到一个 solr-5.3.1.zip 文件，如果你是在 Linux 平台下，那么将得到一个 solr-5.3.1.tgz 文件。这里我首先以 Windows 平台为例，请使用任意解压缩工具，如 WinRAR，直接解压到当前文件夹，解压之后你将看到如图 1-7 所示的目录结构。

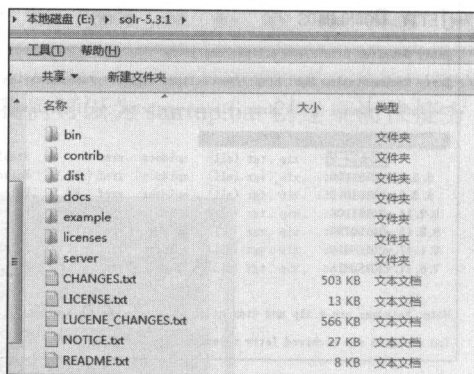


图 1-7 Solr 5.x 解压目录结构

□ bin：这里是官方提供的一些安装部署 Solr 脚本。

□ contrib：这里是社区贡献代码，并未正式纳入 Solr。

□ dist：这里是 Solr 源码编译后打包好的 jar 包。

□ docs：Solr 的 HTML 格式的 API 文档。

□ example：这里是官方提供的 Solr 索引导入 demo。

□ licenses：这里列出了 Solr 遵循的所有开源协议文件。

□ server：这个目录有点意思，取名 server，有点迷惑人，其实就是一个 Jetty。官方为了方便用户快速上手 Solr，在其安装包内内置了一个 Jetty。

□ webapps：这里存放的是 Solr 的部署包 solr.war，这个目录在 Solr 5.x 系列版本中一直都是存在的，不过到 Solr 6.x 版本后，这个目录消失了。其实官方不过是把 war 包解压到 server\solr-webapp 目录下。我们可以直接打 war 包，用命令行下切到 solr-webapp\webapp 目录下，然后执行如下命令：

```
jar -cvf solr.war ./*
```

运行 Solr 最简单的办法就是把 war 包部署到 Jetty 容器里，那么首先我们需要去下载安装 Jetty，关于 Jetty 的更多内容，请参考以下网址的内容，这里不再赘述。

□ Jetty 官网：<http://www.eclipse.org/jetty/>。

□ Jetty 官方文档：<http://www.eclipse.org/jetty/documentation>。

□ Jetty 下载地址：<http://download.eclipse.org/jetty/>。

如图 1-8 所示, stable 有稳定的意思, 表示这是一个稳定版本, tgz 是 Linux 下的压缩包, Windows 下请下载 zip 压缩包, apidocs 是 Jetty 的在线 API 文档, xref 是 Jetty 的在线源码查看地址, 这里我下载的是 stable-9 版本。Jetty 下载下来后, 可以解压到任意盘符, 这里我直接解压到 E 盘, 并更名为 jetty-9.2.10。Jetty 提供了一个 start.jar 来启动 Jetty。如果你的 jar 类型文件默认是自动关联的类似 WinRAR 的解压软件, 那么双击 start.jar 是不能运行 jar 文件的, 需要首先关联 jar 类型文件与 Java 程序, 具体操作如图 1-9 所示。

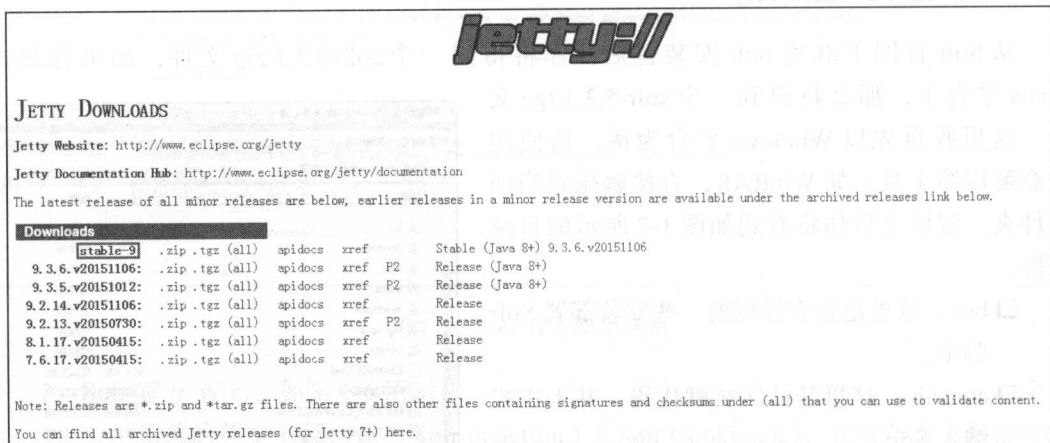


图 1-8 Jetty 官网下载地址

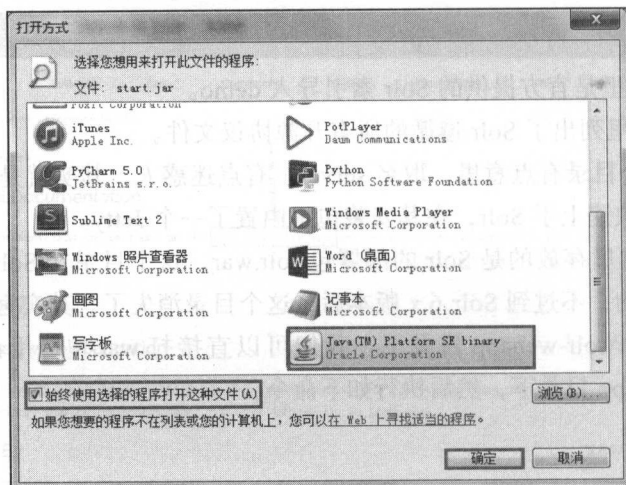


图 1-9 Jar 文件与 Java 程序关联

然后你就可以直接双击 start.jar 来启动 Jetty 了。为了操作方便, 我们也可以新建一个 bat 批处理文件, 如图 1-10 所示。

使用任意一款你喜爱的文本编辑软件打开 startup.bat 文件, 编辑内容如下:

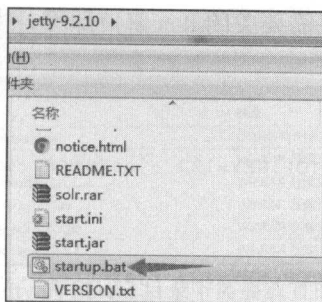


图 1-10 Jetty 启动脚本

```
echo "begin start the jetty....."
java -jar %cd%/start.jar
```

然后双击 bat 文件即可启动 Jetty，为了方便，我们可以为 startup.bat 创建桌面快捷方式，Jetty 启动后如图 1-11 所示。

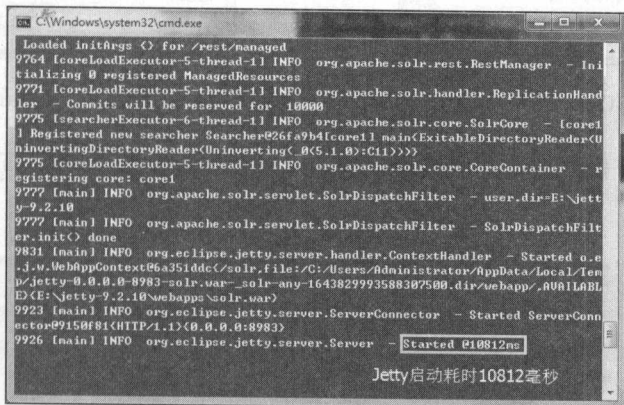


图 1-11 Jetty 启动后效果图

如果你的操作系统开启了防火墙，那么 Jetty 启动过程中可能会弹出如图 1-12 的确认提示框，请选择允许访问。

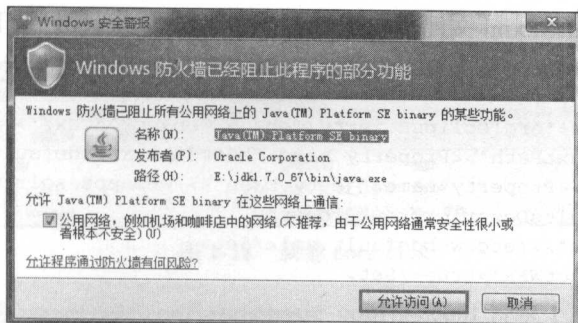


图 1-12 Jetty 启动被系统防火墙拦截示意图

如果需要关闭 Jetty，我们只需要关闭 dos 窗口即可，当然还有另一个方式来启动和关闭 jetty，命令如下：

□ 启动：

```
java -DSTOP.PORT=8009 -DSTOP.KEY=123 -jar start.jar
```

□ 关闭：

```
java -DSTOP.PORT=8009 -DSTOP.KEY=123 -jar start.jar -stop
```

为了方便操作，我们也可以把上面的命令写入 bat 文件，一个 startup.bat 用来启动 Jetty，一个 shutdown.bat 用来关闭 Jetty。然后设置 Jetty 的环境变量，如图 1-13 和图 1-14 所示。

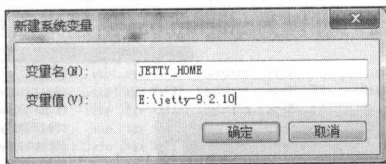


图 1-13 JETTY\_HOME 系统环境变量设置

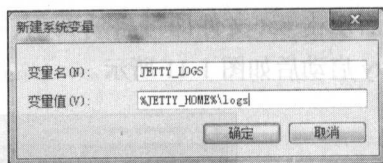


图 1-14 JETTY\_LOGS 系统环境变量设置

这一切准备工作做好后，我们需要把 war 包复制到 Jetty 的 webapps 目录下，如图 1-15 所示：

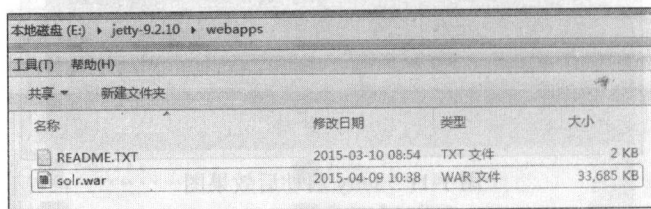


图 1-15 Solr.war 包放置目录示意图

在 Jetty 根目录下新建文件夹 solr-webapp 和 contexts，然后将 E:\solr-5.3.1\server\contexts 目录下的 solr-jetty-context.xml 文件复制至 E:\jetty-9.2.10\contexts 目录下，然后打开 E:\jetty-9.2.10\contexts\solr-jetty-context.xml 对其稍作修改，配置如下代码所示：

```
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
<Set name="contextPath"><Property name="hostContext" default="/solr"/></Set>
<Set name="war"><Property name="jetty.base"/>/webapps/solr.war</Set>
<Set name="defaultsDescriptor"><Property
name="jetty.base"/>/etc/webdefault.xml</Set>
<Set name="extractWAR">true</Set>
</Configure>
```

接着将 server\lib\ext 目录下的所有 jar 包复制到 Jetty 的 lib\ext 目录下，如图 1-16 所示。



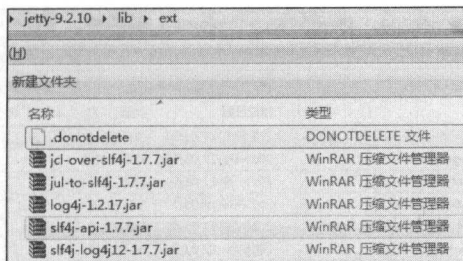


图 1-16 Jetty 的 lib\ext 目录下需要放置的 jar 包截图

然后你需要在 Jetty 根目录下创建一个文件夹作为我们的 solr\_home，如图 1-17 所示。



图 1-17 Jetty 安装根目录新建 SOLR\_HOME 目录

然后复制 solr 的 server\solr 目录下的 solr.xml 文件至刚刚新建的 solr\_home 目录下，并且在 solr\_home 目录下新建一个 core 目录，core 名称命名支持自定义，这里取名为 core1，如图 1-18 所示。

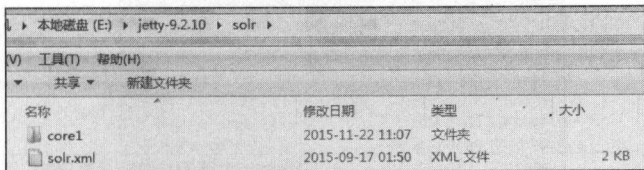


图 1-18 新建 core 目录

紧接着将 solr 的 server\solr\configsets\basic\_configs\conf 目录下的一些配置文件复制到刚刚新建的 core1 的 conf 目录下，如图 1-19 和图 1-20 所示。



图 1-19 复制 conf 下的配置文件

注意：除了 rest manaed.json 文件，其余全部复制。



图 1-20 新建 Core 的 conf 目录示意图

上述准备工作都完成后，双击 Jetty 的 start.bat 文件启动 Jetty，然后打开浏览器输入 <http://localhost:8983/solr> 访问 Solr 的 Web 后台，如果你能看到如图 1-21 所示的界面，那么恭喜你，solr 5.3.1 部署成功了！

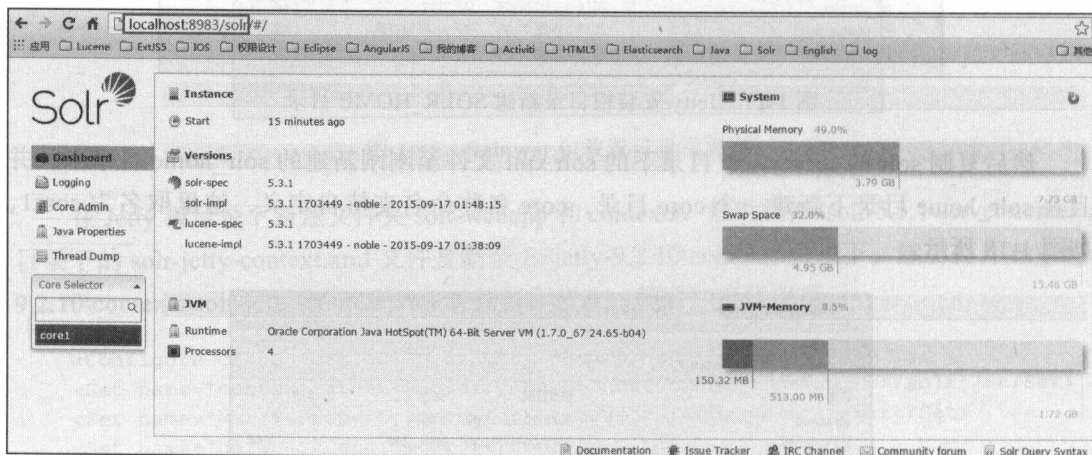


图 1-21 Solr 启动后效果图

Jetty 的默认启动端口号是 8983，你可以通过 `${JETTY_HOME}/start.ini` 配置文件进行修改，具体配置如下所示：



```
# -----
# Module: http
--module=http
### HTTP Connector Configuration
## HTTP port to listen on
jetty.port=8983
## HTTP idle timeout in milliseconds
http.timeout=30000
## HTTP Socket.soLingerTime in seconds. (-1 to disable)
# http.soLingerTime=-1
```



**注意** 修改完端口号后，请重启 Jetty 使配置生效。

## 1.7.2 部署 Solr 至 Tomcat

需要说明的是，这里对 Tomcat 版本没过多要求，你只需要确保 Tomcat 能正常启动。

其实 Solr 在 Tomcat 容器下的部署跟 Jetty 差不多，首先，你需要把我们在上一章节打包好的 solr.war 文件复制到 Tomcat 的 webapps 目录下，如图 1-22 所示。



图 1-22 solr.war 包放置到 Tomcat 的 webapps 目录下

启动你的 Tomcat，如图 1-23 所示。

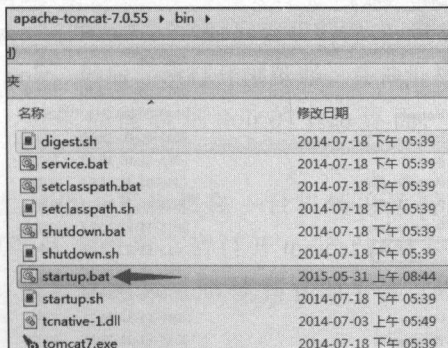


图 1-23 启动 Tomcat Server

然后修改 webapps\solr\WEB-INF 下的 web.xml 配置文件，如图 1-24 所示。

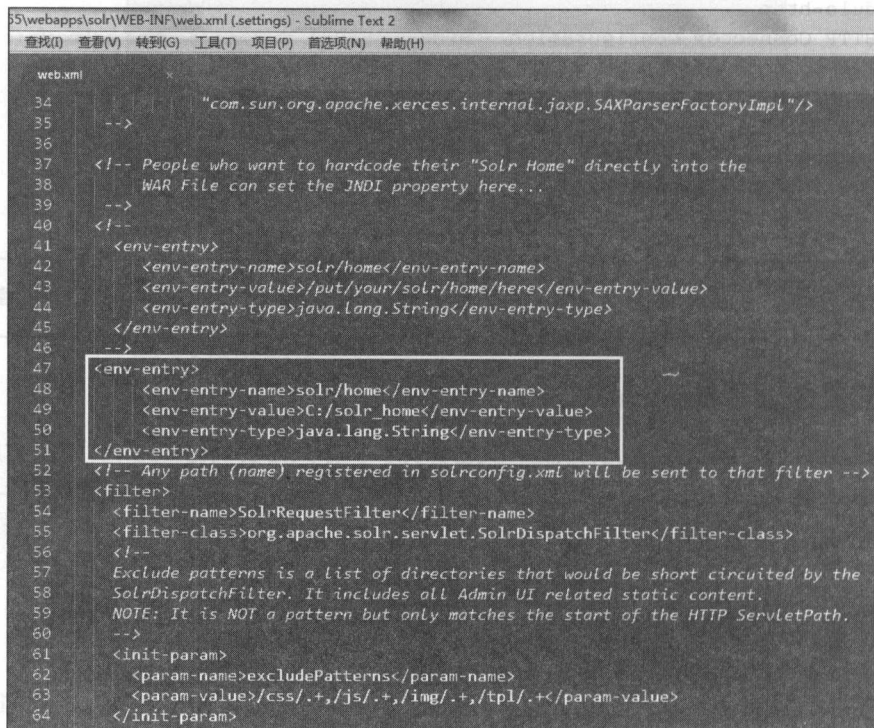


图 1-24 web.xml 配置

这里我把 solr\_home 目录设置为 C:/solr\_home，其实 solr\_home 完全可以自定义。然后我们需要去 C 盘创建一个 solr\_home 文件夹，这就是 SOLR\_HOME 的根目录啦，如图 1-25 所示。



图 1-25 创建 \${SOLR\_HOME} 目录

这里的 solr\_home 目录其实就好比我们在上一章节里在 Jetty 根目录下创建的 solr 目录, 所以为了简便, 可以直接将上一章节里创建的 solr 文件夹下的所有文件及文件夹全部复制到当前 solr\_home 目录下, 然后把 E:\solr-5.3.1\server\lib\ext 目录下的所有 jar 包复制到 E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\lib 目录下, 如图 1-26 所示。

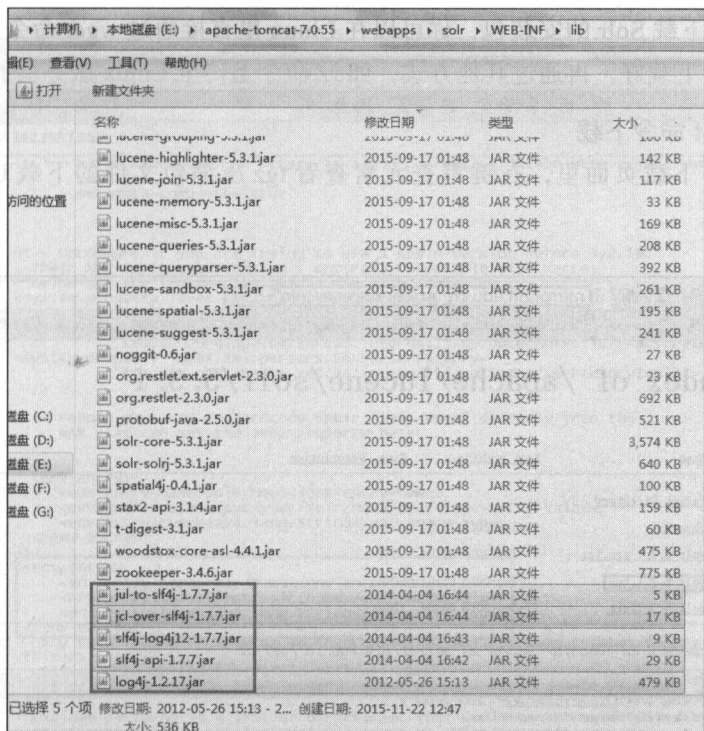


图 1-26 Tomcat 的 lib 目录下依赖的 jar 包截图

接下来, 在 E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF 目录下手动创建一个 classes 文件夹, 用于放置 log4j 配置文件, 并将 E:\solr-5.3.1\server\resources 目录下的 log4j.properties 配置文件复制到新建的 classes 文件夹下, 之后重启我们的 Tomcat, 不出意外的话, Tomcat 会启动成功并且会重新部署 Solr。最后, 请打开浏览器, 在地址栏输入 <http://localhost:8080/solr> 访问 Solr 的 Web 后台, 如果你能看到 Solr 的 Web 界面, 就表明你的 Solr 5.3.1 在 Tomcat 中部署成功了!

Tomcat 的默认端口号是 8080, 当你想在一台机器上运行多个 Tomcat 实例的时候, 有可能需要修改 Tomcat 的默认端口号。请打开 tomcat\conf 目录下的 server.xml 进行编辑, 修改完成后保存退出, 重启你的 Tomcat 使其立即生效。修改 Tomcat 的端口号示例如下:

```
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
```

## 1.8 Linux 平台下部署 Solr

假定你有一台 Linux 服务器且已经安装好 JDK 7+ 以及 Tomcat, 这些运行环境的搭建不属于本书的主题, 所以这里不做过多说明。关于这方面的资料请查阅相关资料。这里我以 CentOS 6.5 为例进行讲解。

首先, 需要下载 Solr 的安装包, 你可以在 Linux 里直接通过 `wget` 命令下载, 也可以先在 Windows 平台下载好, 再通过其他方式, 如 `rz` 命令上传至 Linux 服务器的指定目录下。

### 1. 通过 `wget` 命令下载

在 Solr 官方下载页面里, 右键审查元素查看 `tgz` 压缩包文件的下载 URL, 如图 1-27 所示。

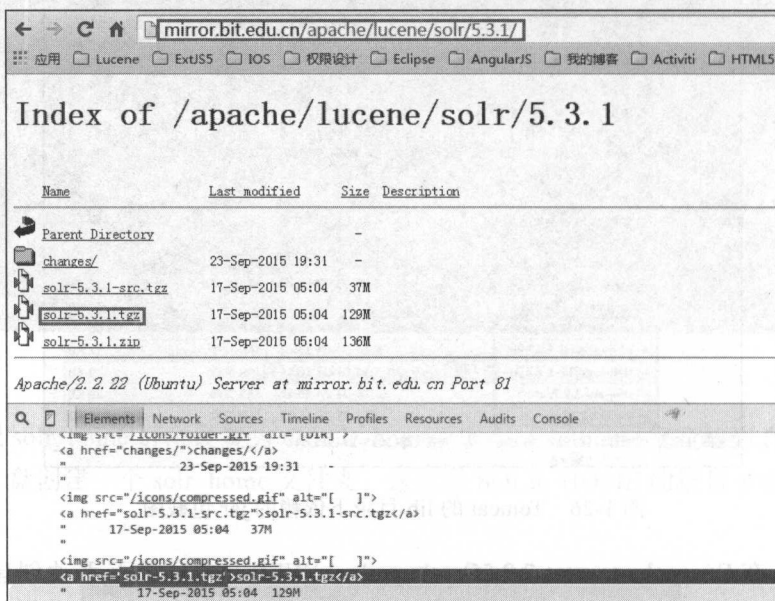


图 1-27 获取 solr 下载 URL

这里的 URL 是相对路径, 是相对当前页面的访问 URL, 因此最后完整的下载 URL 是: `http://mirror.bit.edu.cn/apache/lucene/solr/5.3.1/solr-5.3.1.tgz`, 知道了如何下载 URL, 那接下来就可以在 Linux 里通过 `wget` 直接下载了, 命令如下:

```
wget http://mirror.bit.edu.cn/apache/lucene/solr/5.3.1/solr-5.3.1.tgz
```

### 2. 在 Windows 平台下载再上传到 Linux

可以使用 `rz` 命令进行文件上传, 但我建议使用 Xftp 工具进行文件上传下载操作, 至于 Xftp 工具如何下载安装以及如何操作使用, 请自行查询并学习, 这里不做说明。

上传完成后你需要使用 tar 命令解压缩 solr-5.3.1.tgz,

```
tar -zxvf solr-5.3.1.tgz
```

然后需要在 solr 的 web.xml 里配置 solr\_home 目录, 使用 vi 对 /opt/solr-5.3.1/server/solr-webapp/webapp/WEB-INF 下的 web.xml 进行编辑, 如图 1-28 所示。

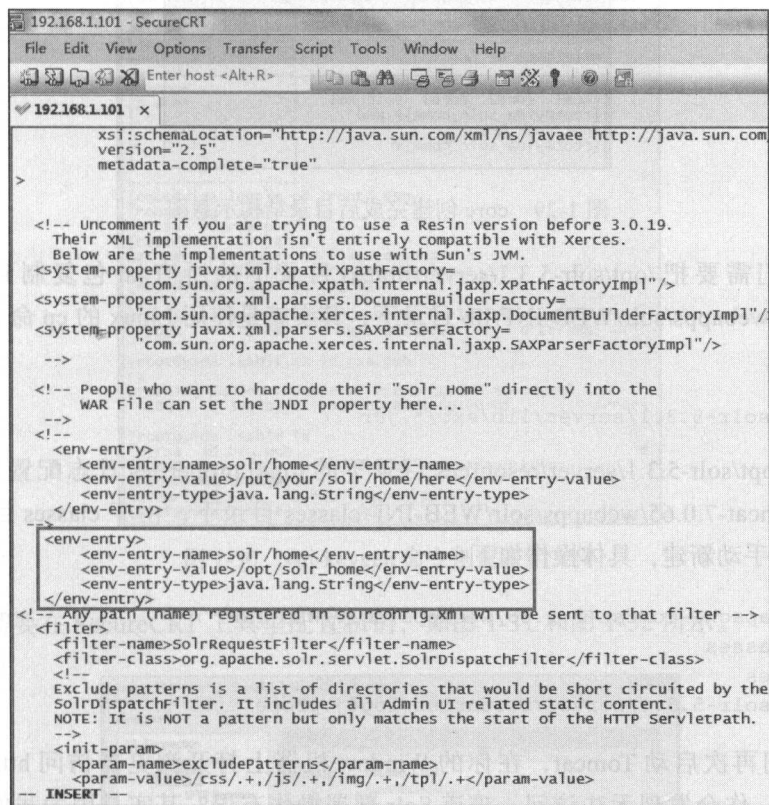


图 1-28 使用 vi 编辑 web.xml

修改完成后保存退出, 我们需要将其打成 war 包方便部署, 先将当前目录切换至 /opt/solr-5.3.1/server/solr-webapp/webapp, 然后使用 zip 命名进行打包压缩, 使用命令如下:

```
zip -r solr.war ./*
```

在当前路径下会生成一个 solr.war 压缩包, 紧接着需要将其复制到 Tomcat 的 webapps 目录下。这样就将 solr 部署到 Tomcat 了, 但目前还不能正常访问, 因为 solr\_home 目录还未创建。solr\_home 下还需要创建一个 core, 并需要在 core 的 conf 目录下放置必需的配置文件, 如图 1-29 所示, 这些在前面已经做过说明, 为了节省篇幅, 这里直接将 Windows 平台下创建的 solr\_home 目录压缩成 zip 上传至 Linux 服务器。



```
unzip -o solr_home.zip // 解压缩上传的 solr_home.zip 文件
```

这样我们就把在 Windows 上创建的 3 个测试 core 复制到了 Linux 系统上，算是偷了个懒。不会偷懒的程序员不是一个好程序员，哈哈！

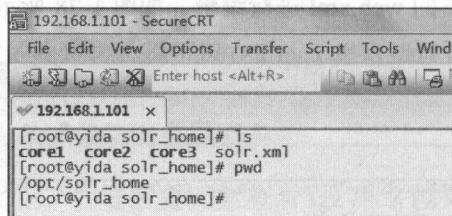


图 1-29 core 创建完成后目录结构示意图

接着我们需要把 /opt/solr-5.3.1/server/lib/ext 目录下的所有 jar 包复制到 /opt/apache-tomcat-7.0.65/webapps/solr/WEB-INF/lib 目录下，其实就是使用 Linux 的 cp 命令进行文件复制操作。

```
cp /opt/solr-5.3.1/server/lib/ext/*.jar ./
```

然后将 /opt/solr-5.3.1/server/resources 目录下的 log4j.properties 日志配置文件复制到 /opt/apache-tomcat-7.0.65/webapps/solr/WEB-INF/classes 目录下，由于 classes 目录默认不存在，我们需要手动新建，具体操作如下：

```
cd /opt/apache-tomcat-7.0.65/webapps/solr/WEB-INF/  
mkdir classes  
cd classes  
cp /opt/solr-5.3.1/server/resources/log4j.properties ./
```

最后我们再次启动 Tomcat，在你的 Window 机器上打开浏览器访问 <http://192.168.1.101:8080/solr>，你会发现无法访问。难道 Solr 部署操作有误？其实是因为我们的 Windows 机器并不能直接访问 Linux，需要先用账号密码登录 Linux 系统才可以。所以我们还需要设置 Windows 免密码登录 Linux。具体操作如下：

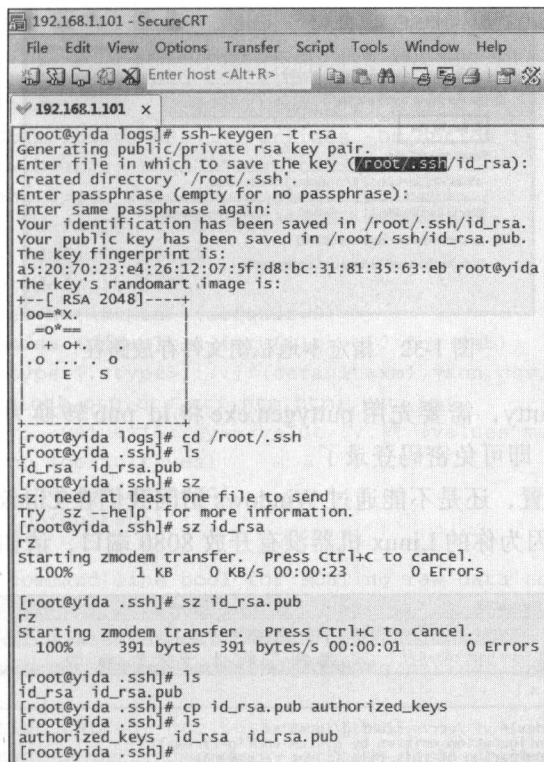
首先用 root 登录 Linux，运行 `ssh-keygen -t rsa`，按 3 次回车，生成 2 个文件：

- /root/.ssh/id\_rsa 私钥；
- /root/.ssh/id\_rsa.pub 公钥。

将这两个文件通过 `sz` 命令传至 Windows 机器，比如 c:\key 目录下（该目录需要你提前创建好）。

```
cd /root/.ssh // 切换到 /root/.ssh 目录下  
cp id_rsa.pub authorized_keys // 复制 id_rsa.pub 文件并命名为 authorized_keys
```

具体操作如图 1-30 所示。



```

192.168.1.101 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+R>
192.168.1.101 x
[root@yida logs]# ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
a5:20:70:23:e4:26:12:07:5f:d8:bc:31:81:35:63:eb root@yida
The key's randomart image is:
+---[ RSA 2048]-----+
|  oo=X.                                     |
|  =O*=+                                     |
|  o + O+.                                  |
|  .O ... .O                                |
|      E  S                                 |
+-----+
[root@yida logs]# cd /root/.ssh
[root@yida .ssh]# ls
id_rsa  id_rsa.pub
[root@yida .ssh]# sz
sz: need at least one file to send
Try 'sz --help' for more information.
[root@yida .ssh]# sz id_rsa
rz
Starting zmodem transfer. Press Ctrl+C to cancel.
100% 1 KB 0 KB/s 00:00:23 0 Errors
[root@yida .ssh]# sz id_rsa.pub
rz
Starting zmodem transfer. Press Ctrl+C to cancel.
100% 391 bytes 391 bytes/s 00:00:01 0 Errors
[root@yida .ssh]# ls
id_rsa  id_rsa.pub
[root@yida .ssh]# cp id_rsa.pub authorized_keys
[root@yida .ssh]# ls
authorized_keys  id_rsa  id_rsa.pub
[root@yida .ssh]#

```

图 1-30 ssh-keygen 命令创建密钥文件

然后你需要在 SecureCRT 工具里配置私钥，如图 1-31 和图 1-32 所示。

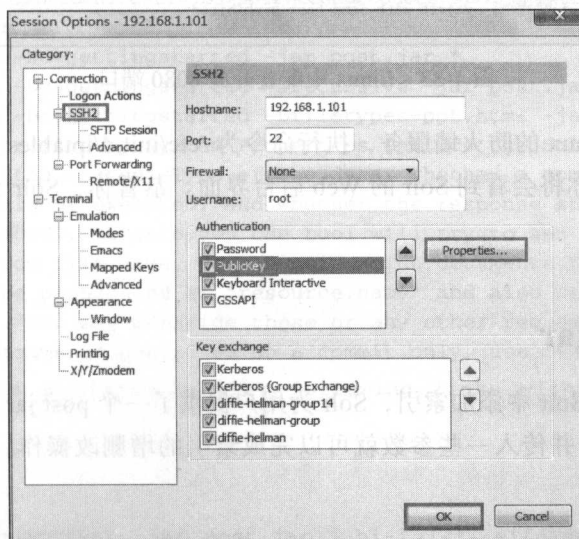


图 1-31 SecureCRT 里配置私钥

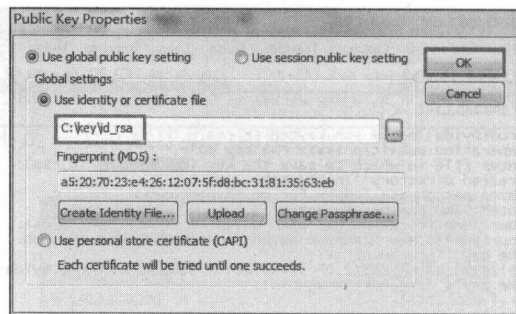


图 1-32 指定本地私钥文件存放路径

如果你使用的是 putty, 需要先用 puttygen.exe 将 id\_pub 转换为 id\_pub.ppk。设置 ssh/auth, 选择 id\_pub.ppk, 即可免密码登录了。

如果设置了上述配置, 还是不能通过 Windows 访问我们在 Linux 机器 192.168.1.101 上部署的 Solr, 那可能是因为你的 Linux 机器没有开放 8080 端口, 请如图 1-33 所示操作。

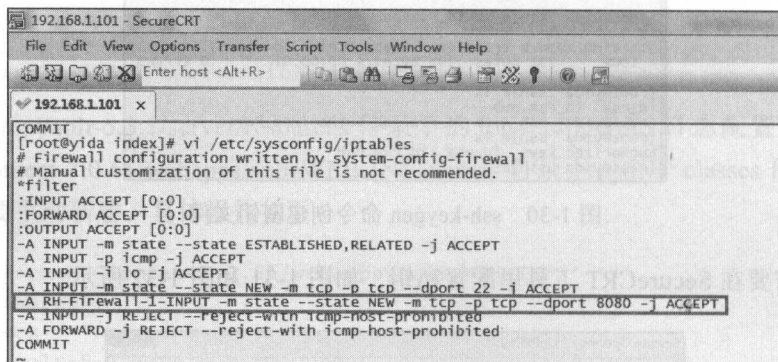


图 1-33 Linux 中配置开放 8080 端口

最后需要重启 Linux 的防火墙服务, 执行命令为 `/etc/init.d/iptables restart`。重启完成后, 请再次刷新浏览器, 你将会看到 Solr 的 Web 后台界面。恭喜你, Solr 5.3.1 在 Linux 服务器上就部署成功啦!

## 1.9 玩转 post.jar

为了方便用户向 Solr 中添加索引, Solr 为用户提供了一个 post.jar 工具, 用户只需要在命令行下运行 post.jar 并传入一些参数就可以完成索引的增删改操作。有关 post.jar 的使用说明如下:

```
SimplePostTool version 5.1.0
```

```
Usage: java [SystemProperties] -jar post.jar [-hl-] [<file|folder|url|arg> [<file|
```



folder|url|arg>...]]

Supported System Properties and their defaults:

- Dc=<core/collection>
- Durl=<base Solr update URL> (overrides -Dc option if specified)
- Ddata=files|web|args|stdin (default=files)
- Dtype=<content-type> (default=application/xml)
- Dhost=<host> (default: localhost)
- Dport=<port> (default: 8983)
- Dauto=yes|no (default=no)
- Drecursive=yes|no|<depth> (default=0)
- Ddelay=<seconds> (default=0 for files, 10 for web)
- Dfiletypes=<type>[,<type>,...] (default=xml,json,csv,pdf,doc,docx,ppt,pptx,xls,xlsx,odt,ods,ott,otp,ots,rtf,htm,html,txt,log)
- Dparams="<key>=<value>[&<key>=<value>...]" (values must be URL-encoded)
- Dcommit=yes|no (default=yes)
- Doptimize=yes|no (default=no)
- Dout=yes|no (default=no)

This is a simple command line tool for POSTing raw data to a Solr port.

NOTE: Specifying the url/core/collection name is mandatory.

Data can be read from files specified as commandline args,

URLs specified as args, as raw commandline arg strings or via STDIN.

Examples:

```
java -Dc=gettingstarted -jar post.jar *.xml
java -Ddata=args -Dc=gettingstarted -jar post.jar '<delete><id>42</id></delete>'
java -Ddata=stdin -Dc=gettingstarted -jar post.jar < hd.xml
java -Ddata=web -Dc=gettingstarted -jar post.jar http://example.com/
java -Dtype=text/csv -Dc=gettingstarted -jar post.jar *.csv
java -Dtype=application/json -Dc=gettingstarted -jar post.jar *.json
java -Durl=http://localhost:8983/solr/techproducts/update/extract -Dparams=literal.
id=pdf1 -jar post.jar solr-word.pdf
java -Dauto -Dc=gettingstarted -jar post.jar *
java -Dauto -Dc=gettingstarted -Drecursive -jar post.jar afolder
java -Dauto -Dc=gettingstarted -Dfiletypes=ppt,html -jar post.jar afolder
```

The options controlled by System Properties include the Solr URL to POST to, the Content-Type of the data, whether a commit or optimize should be executed, and whether the response should be written to STDOUT. If auto=yes the tool will try to set type automatically from file name. When posting rich documents the file name will be propagated as "resource.name" and also used as "literal.id". You may override these or any other request parameter through the -Dparams property. To do a commit only, use "-" as argument.

The web mode is a simple crawler following links within domain, default delay=10s.

### 重点代码解析:

```
java [SystemProperties] -jar post.jar [-h|-] [<file|folder|url|arg> [<file|
folder|url|arg>...]]
```

要看懂这个 `post.jar` 使用命令规范，你首先需要知道，被中括号包住的参数为可选参数，“|”表示或者，`SystemProperties` 表示系统属性，即你通过 `System.setProperty()`；设置的参数，比如：

```
System.setProperty(key,value);
```

这里的 `key`、`value` 值都是随便定义的，你可以通过 `System.getProperty(key)` 在任意时刻获取到该 `key` 对应的参数值，如果是在 `dos` 命令行下，也可以通过 `java -Dkey=value` 这种方式指定，至此 `java [SystemProperties]` 这部分你应该理解了。后面的 `-jar` 是 `java` 命令的参数，即执行一个 `jar` 文件，`-jar` 后面指定一个 `jar` 包路径，默认是相对于当前所在路径，`-h` 表示打印命令提示信息，与你敲 `java -h` 是类似的。后面的 `file`、`folder`、`url`、`args` 分别表示要提交数据的几种不同表示形式：`file` 表示你要提交的数据是存在于文件中；`folder` 表示你要提交的存在于文件夹中；`url` 表示你要提交的数据是存在于互联网上的一个 `URL` 地址表示的资源，它可能是一个 `HTML` 页面，可能是一个 `PDF` 文件，也可能是一个图片等；`args` 表示你要提交的数据直接在命令行敲出来。`args` 并不是随随便便一个字符串就行的，它需要有固定的格式，才能让 `Solr` 解析，其输入格式后面会说到。

下面列出 `post.jar` 支持的几个自定义系统属性，后续我会对每个自定义系统属性一一说明：

```
-Dc=<core/collection>
-Durl=<base Solr update URL> (overrides -Dc option if specified)
-Ddata=files|web|args|stdin (default=files)
-Dtype=<content-type> (default=application/xml)
-Dhost=<host> (default: localhost)
-Dport=<port> (default: 8983)
-Dauto=yes|no (default=no)
-Drecursive=yes|no|<depth> (default=0)
-Ddelay=<seconds> (default=0 for files, 10 for web)
-Dfiletypes=<type>[,<type>,...] (default=xml,json,csv,pdf,doc,docx,ppt,pptx,xls,xlsx,odt,odp,ods,ott,otp,ots,rtf,htm,html,txt,log)
-Dparams="<key>=<value>[&<key>=<value>...]" (values must be URL-encoded)
-Dcommit=yes|no (default=yes)
-Doptimize=yes|no (default=no)
-Dout=yes|no (default=no)
```

❑ `-D`：命令行下指定系统属性的固定前缀；

❑ `c`：Core 名称，即你需要对 `Solr Admin` 里的哪个 Core 进行相关操作；

❑ `url`：表示 `Solr Admin` 后台索引更新的请求 `URL`，这个 `URL` 是固定的，一般格式是 `http://host:port/solr/${coreName}/update`，这里的 `${coreName}` 和上面的 `c` 属性值保持一致；

❑ `data`：表示要提交数据的几种模式，其中，`files` 模式表示你要提交的数据在文件里；

❑ `web`：表示要提交的数据在互联网上的一个 `URL` 表示的资源文件里；

❑ `args`：表示要提交的数据会在 `post.jar` 命令后面直接输入；

- **stdin** : 表示要提交的数据需要在 dos 命令行下通过 System.in 输入流临时接收, 跟 args 有点类似, 但不同的是, stdin 模式下, post.jar 后面不需要指定任何参数, 直接回车即可, 然后程序会等待用户输入, 用户输入完毕再回车, post.jar 会接收到用户输入, 并重新被唤醒继续执行。而 args 是直接 post.jar 后面输入参数, 没有一个中断过程, 而 stdin 模式下如果用户一直没有输入, 那 post.jar 就会一直阻塞在那里等待直到用户输入为止;
- **type** : 表示要提交数据的 MIME 类型, 默认是 application/xml 即默认会当作是 XML 来处理;
- **host**: 表示要连接的 Solr Admin 部署服务器的主机名或者 IP 地址, 默认是 localhost;
- **port**: 表示要连接的 Solr Admin 部署的 Web 容器监听的端口号, 默认 post.jar 里设置为 8983, port 具体值取决于你实际的部署环境;
- **auto**: 表示是否自动猜测文件类型;
- **recursive** : 表示是否递归, 这里递归有两种情况, 比如你 data=folder 即表示是否递归查找文件夹下的所有文件, 如果你 data=web 即表示是否递归抓取 URL, 设置为 no 即表示不递归操作, 设置为一个数字, 即表示递归深度;
- **delay** : 这里的时间延迟也分两种, 如果你 post 的是 file, 那么每个 file 的 post 间隔为 0, 即不做延迟处理, 而如果你 post 的是网络上的一个 url 资源, 因为需要收到对方服务器的访问限制, 所以必须要做一个抓取频率限制即每抓一个睡眠一会儿, 否则抓取太快频率太高容易被对方封 IP;
- **filetypes** : 表示 post.jar 支持提交哪些文件类型, 后面有列出默认支持的文件类型, 如果你想覆盖默认值, 那么请指定此参数;
- **params**: 表示需要追加到 Solr Admin 的请求 URL 后面的参数如 id=1&name=yida 之类的;
- **commit**: 表示是否提交到 Solr Admin 后台进行索引写入, 设置为 false 表示不提交至 Solr Admin, 但设置为 true 也不一定就意味着就一定会把索引写入磁盘, 这取决于 solrconfig 中 directory 配置的实现是什么, 如果配置的是 RAMDirectory, 就仅仅只在内存中操作了;
- **optimize**: 表示是否需要索引进行优化操作, 默认为 no, 即不对索引进行优化;
- **out**: 即 OutputStream 表示输出流, 这个参数作用就是, 你请求 Solr Admin 添加索引数据, Solr Admin 后台会返回数据给你, Solr Admin 后台返回的数据你拿什么输出流来接收, 默认是 System.out 即表示把后台返回的信息输出打印到控制台;

理解上面的相关说明后, 再来看看官方提供的几个 post.jar 使用命令示例, 是不是感觉很简单了?

Examples:

```
java -Dc=gettingstarted -jar post.jar *.xml
java -Ddata=args -Dc=gettingstarted -jar post.jar '<delete><id>42</id></delete>'
```

```

java -Ddata=stdin -Dc=gettingstarted -jar post.jar < hd.xml
java -Ddata=web -Dc=gettingstarted -jar post.jar http://example.com/
java -Dtype=text/csv -Dc=gettingstarted -jar post.jar *.csv
java -Dtype=application/json -Dc=gettingstarted -jar post.jar *.json
java -Durl=http://localhost:8983/solr/techproducts/update/extract -Dparams=literal.id=pdf1 -jar post.jar solr-word.pdf
java -Dauto -Dc=gettingstarted -jar post.jar *
java -Dauto -Dc=gettingstarted -Drecursive -jar post.jar afolder
java -Dauto -Dc=gettingstarted -Dfiletypes=ppt,html -jar post.jar afolder

```

OK, 知道 post.jar 怎么玩了, 那是不是该来实践一把? 要想往 Solr Admin 后台添加索引数据, 你首先需要添加一个 Core, 可以通过 Solr Admin 的 web UI 来创建, 如图 1-34 所示。

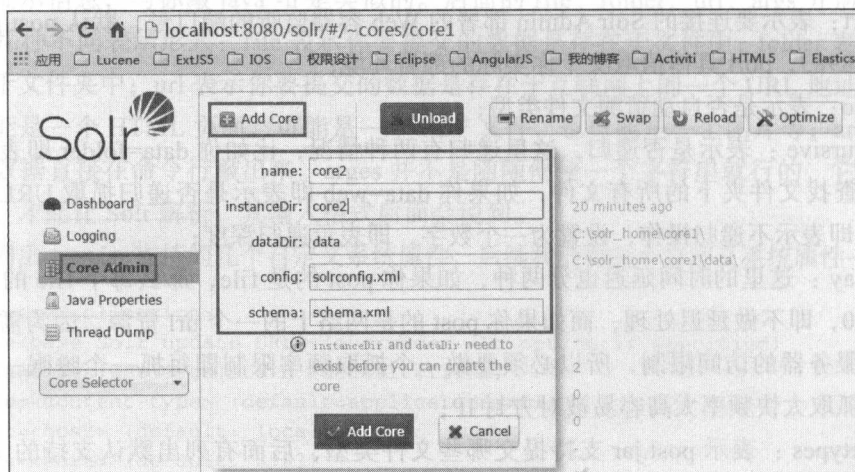


图 1-34 创建 core 示意图

name 即你的 Core 名称, instanceDir 就是你的 Core 根目录, dataDir 表示你 Core 的数据目录, 当前 Core 的索引数据会存放在 dataDir 下的 data/index 目录下, Core 的日志文件则会自动存放在 dataDir 的 data/tlog 目录下。上述所有文件夹需要你手动创建 (除了 data/index 这里的 index 目录, Solr 会自动创建), 至于一些 Core 必需的配置如 solrconfig.xml、schema.xml, 则可以从其他已经配置好的 Core 目录下复制, 最简单的做法是直接复制一份 core1 重命名为 core2, 然后修改 core2 的 core.properties 配置文件里的 name=core2 即可, 创建好的 core2 目录结构如图 1-35 所示。

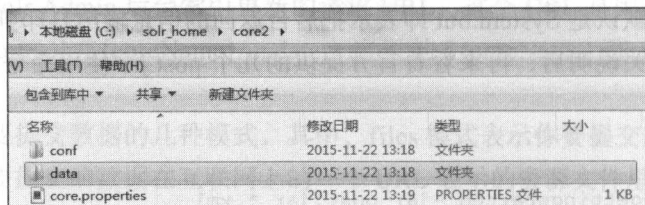


图 1-35 core 目录结构示意图

在添加 Add Core 按钮提交创建 Core 请求之前, 确保你已经创建好 Core 目录及其必需的配置。注意 Core 名称不一定要和你的 Core 目录名词保持一致, Core 目录名称由你的 instanceDir 值决定。有创建 Core, 自然有删除 Core。删除 Core 时, 先选中你需要删除的 Core, 然后单击红色的 Unload 按钮即可。这里说的 Core 删除并不会真实地删除你硬盘上的 Core 目录, 只是让该 Core 不再受 Solr 管理。如图 1-36 所示。

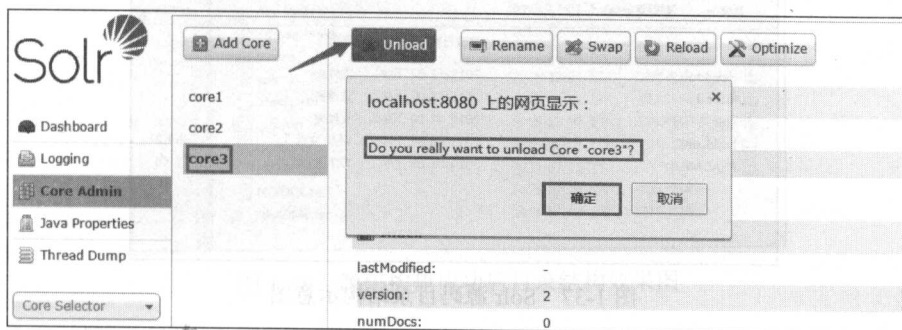


图 1-36 删除 core 操作示意图

- Rename 即修改 Core 名称, 这个功能很简单, 你们自己动手实践去感受下;
- Swap 即动态替换 Core, 即将线上一个正在运行的 Core 用一个替补的 Core 进行替换, 同时保持被替换的 Core 继续运行以便能随时将其回滚。这有点类似于足球比赛里的中途换人。整个替换过程中你的 Solr 服务不需要中断;
- Reload 即重新加载你的 Core, 当修改了 Solr 配置文件时需要使其立即生效, 但这并不意味着你对配置文件做的所有修改能重新支持 Reload 生效, 比如:
  - dataDir 目录不支持 Reload 方式修改;
  - solrconfig.xml 里 <indexConfig> 之间有关 IndexWrite 实例的配置项是不支持 Reload 方式修改的;
  - Optimize 即对指定 Core 的索引数据进行合并优化。

## 1.10 在 Eclipse 中编译 Solr 源码

开始之前你需要准备好构建环境:

- 1) 首先你需要安装 ant 并配置好 ant 的环境变量 ANT\_HOME;
- 2) 下载 IVY 并将 ivy.jar 复制到 ANT\_HOME\lib 目录下;
- 3) 下载并安装 Maven 以及配置好 MAVEN\_HOME 环境变量, 如果有必要的话, 你还可以设置修改下默认的本地仓库位置;
- 4) 你需要下载一个 Eclipse, 由于 Solr 5.x 对于 JDK 要求至少 1.7+, 所以你的 Eclipse 版本最好为 4.x, 否则你的 Eclipse 无法支持 JDK 1.7 进行编译。



特别注意的是 ant 版本最好是 1.8.2+, 以及 ivy 下载完成后, 解压 ivy 压缩包, 然后复制 ivy.jar 包到你的 ANT\_HOME\lib 目录下。之前我们已经下载了 Solr 的源码包, 请直接将其解压至任意目录, 解压后目录结构如图 1-37 所示。

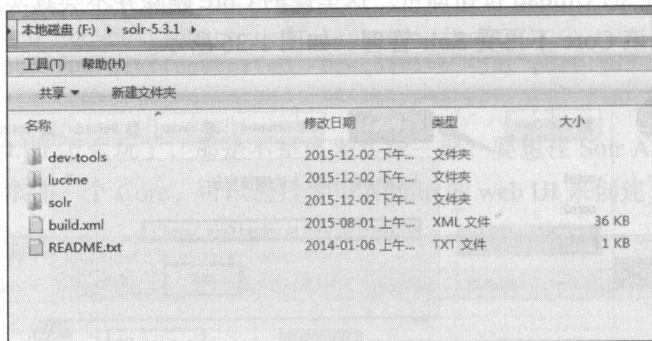


图 1-37 Solr 源码目录结构示意图

重点在解压后根目录下的 build.xml, 建议大家开始编译 Solr 源码之前先熟读 build.xml, 了解下其中到底提供了哪些 ant 自定义 Target。这涉及一些 ant 基础知识, 请自行查阅。

通过阅读 build.xml, 我们可以了解到要想将 Solr 源码导入 Eclipse 需要通过 ant eclipse 命令进行转换, 为了方便看到异常信息, 建议加上 verbose 参数, 即 ant eclipse -verbose, 这样有助于排错。转换过程耗时长短取决于你的网络环境, 过程中可能会遇到异常, 通常是提示某个 jar 依赖找不到, 这时你需要根据异常提示, 到 ivy 的默认本地仓库地址 C:\Users\Administrator\ivy2\cache (这里以 Win7 为例) 找到下载失败的 jar 的所在目录, 将其删除然后再次执行 ant eclipse 命令, 如果中途仍出现异常, 请重复上述步骤直到源码转换构建成功。图 1-38 是我构建成功后的截图。

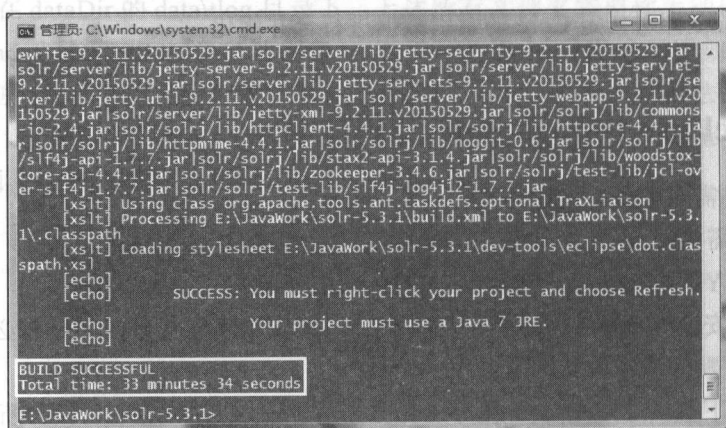


图 1-38 Solr 源码编译成功示意图



源码构建成功后的目录结构如图 1-39 所示。

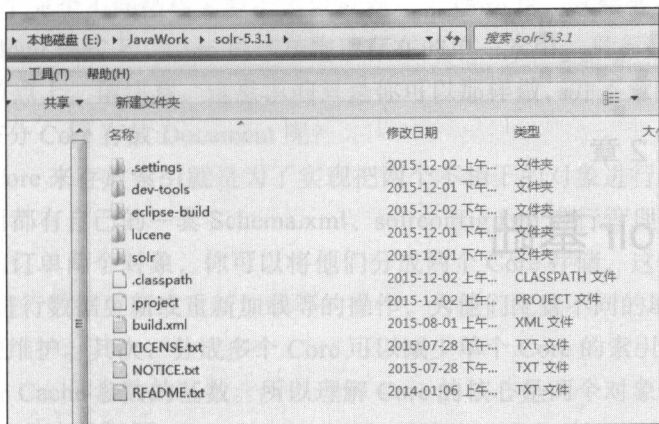


图 1-39 源码构建成功后目录结构效果图

最后，你就可以将其当作一个普通的 Java 项目导入到你的 Eclipse 中啦。至于如何构建 Solr 的源码并导入 IDEA 中，步骤大致相同，唯一区别就是要将 ant eclipse 命令改成 ant idea，这个留给大家亲自实践。

Solr 源码导入 Eclipse 中可以方便查阅，比在常见的文本编辑软件里要方便许多，但是目前还不能对源码进行断点调试。因为目前你导入的仅仅是一个普通的 Java 项目，需要把它转换成 Web 项目才可以，即把 Solr 源码和 Solr Web UI 整合到一起，Solr Web UI 部分代码在 solr.war 里解压可以得到。至于具体如何操作，由于太费篇幅，你可以参考我上传到 Github 上整合的目录结构，附上 solr-5.3.1-web 的 github 下载地址：<https://github.com/yida-lxw/solr-5.3.1-web>。

## 1.11 本章总结

在本章中，我们首先宏观上了解了 Solr 到底是什么并简单回顾了 Solr 的发展历史。紧接着我们了解了为什么需要选择 Solr。在进行技术选型时，“为什么要选择 Solr”是你需要回复你上级的关键问题。如果你自己本身就是 TeamLeader，那么你更需要了解这个问题，从而权衡技术选型的利弊。然后我们简单了解了 Solr 目前已经提供给用户的功能，快速知晓了 Solr 能帮我们做什么。学习一门技术，离不开相关学习资源。如果你英语阅读水平过关，我建议你直接阅读官方技术文档。即便你英语水平一般，我也强烈建议你现在就养成阅读英文技术文档的习惯。最后我们学习了如何在 Windows 和 Linux 平台下部署 Solr，以及如何使用 Solr 官方提供的 post.jar 工具。如果想要阅读 Solr 的源码，那么你可以选择性地阅读 1.10 小节。如果你使用的是 IDEA 开发工具，也可以触类旁通。

## Solr 基础

通过第 2 章，你将可以学习到如下内容：

- ☐ 理解 Solr Core 的基本概念；
- ☐ 掌握 Solr Core 的基本管理；
- ☐ 掌握如何使用 Solr DataImporter 导入数据；
- ☐ 掌握 Solr Cell；
- ☐ 了解 SolrDuplication Detection（索引去重检测）；
- ☐ 了解 Luke 的基本使用。

### 2.1 Solr Core

我们的索引数据一般都是基于 Core 进行组织管理的，自然离不开对 Core 的一些管理操作，比如新建 Core、卸载 Core、重新加载 Core、重命名 Core、Core 热交换、Core 优化、导入数据到 Core 中、Core 之间的索引数据合并等。

#### 2.1.1 Solr Core 简介

想要在 Solr 中添加索引，你需要指定一个 Core，即你需要把索引数据添加到哪个 Core 中。那么问题来了，Core 是什么？我想这是初学 Solr 的同学最大的疑问。

Solr 中的 Core 术语指的是一个单一的索引数据，而索引又是由多个 Document 组成的，所以你把 Core 理解为多个 Document 打包成的一个集合，就跟超市里卖水果，有散装称斤卖的，也有用保鲜膜或纸箱打包好卖的。需要注意的是，Solr 里的 Document 是扁平化的，

即两个 Document 的域可以完全不同,也就是说表示客户信息的 Document 和表示产品信息的 Document 这两个完全不同结构的数据可以放到一个 Core 里,这跟数据库里的表概念有点不同。在 SolrCloud 模式下,Core 仍然是物理存在的单一索引,只不过不同 Core 可能分布在集群的不同节点上。请注意,这里说的只是你可以那样做,并不意味着推荐你去这么做。那什么时候该分 Core 存放 Document 呢?

Solr 设计多 Core 来存放索引就是为了实现把两个不相干的对象进行分离独立管理,这样每个对象的索引都有自己的一套 Schema.xml、solrconfig.xml 进行管理,彼此之间互不影响。比如有客户和订单两个对象,你可以将他们分成两个 Core 存储,这样能实现单独对其中任意一个 Core 进行数据更新或重新加载等的操作,为他们配置不同的域及其他配置参数,而互不干扰且方便维护。其次,分成多个 Core 可以减少单个 Core 的索引体积大小,这样也影响了你配置 Solr Cache 参数的基数。所以理解 Core 的核心是两个对象是否相干。但同时也要考虑 Core 可能带来的问题:

分成多个 Core,那你可能会遇到跨 Core 的联合查询问题,虽然 Solr 支持 join 来联合多个 Core 进行查询,就好比 SQL 里的 join 查询,但这是有性能损耗的,所以你需要好好权衡。

比如你将学生和老师两个实体分成两个 Core 存放,当删除一个老师对象时,你需要更新  $N$  个关联这个老师的学生对象,这就存在类似 JDBC 里的  $N+1$  问题,即需要  $N+1$  个 HTTP 连接,但如果把两者放到一个 Core 里存放就可以实现发起单个请求解决  $N+1$  问题,你的查询性能也提升了。这不意味着任何时候都优先使用单 Core,需要你在查询性能和系统设计解耦以及系统后期索引数据维护上做个权衡。

再比如你有 Book 和 Shoe 两个互不相干的对象,按照理论来说,需要分 Core 来存放其索引数据。毫无疑问两者都有相同的 price 域,假定你需要查询出价格在  $[X, Y]$  区间内的 Book 和 Shoe,如果你设计成单个 Core,就可以避免跨 Core 查询了。再者,FacetQuery 在跨 Core 下也不会保证正确返回。

很多初学者很容易将 Solr 里的 Core 与数据库领域的 DataBase 做类比来理解,把 Document 类比成数据库里的 Table 来理解,这也无可厚非,但不意味着两者是等同的。在解释 Solr 的 Core 跟 DataBase 的区别之前,我们首先要了解 Solr 能做什么而 Database 做不到的。关系型数据库和 Solr 各有所长。数据库里有简单的基于通配符的文本模糊查询,但这会导致全表扫描,性能很差,而 Solr 是把搜索关键字保存在一个倒排表里,搜索性能提高了  $N$  个数量级。但 Solr 创建索引速度相对较慢。Solr 里更新部分字段(域)数据相对较慢,因为 Solr 里更新只能先删除再新增。而且在新增数据的可见性方面,数据库能立马可见, Solr 近实时查询的数据可见性则稍差些。Solr 的魅力在于它灵活的 Schema 机制,由于 Schema.xml 约束比较宽松,你甚至可以认为 Solr 的 Schema.xml 只是个摆设,每个 Document 可以有任意个任意类型的域,而数据库里的表的字段是提前限定的,且每一行记录拥有的字段数必须一致。了解了这些,我们继续回到最初提出的问题:是不是可以直接用

数据库设计表的那套思路来设计我们 Solr 里的 Core 呢？答案是这取决于很多因素，需要综合考虑。比如你有课程表和分数表数据，你需要从以下几个方面来考虑：

- 1) 你的系统是否明确划分为课程和分数管理两个模块。
- 2) 是否有这样一个场景：你需要返回课程和分数两个类型的全部数据？因为跨 Core Join 查询会损耗性能，这部分损耗你是否可以接受？
- 3) 每个类型索引数据的更新频率一致吗？如果两者其中一个几乎不更新，而另一个经常更新，更新频率不一致，自然不适合混合在一起存为单个 Core。
- 4) 两者数据是否明确做权限划分，即可能希望课程表数据对于 A 应用可见，而对于 B 应用不可见，那你最好是分 Core 存放。
- 5) 分成多个 Core 来存放，带来索引数据维护的复杂度，但分成单 Core 带来索引体积增大，单次索引重建耗时会长。比如 50 000 条鞋子数据和 500 万条衣服数据肯定不适合单个 Core 存放，因为两者不是一个数量级的，放一起会为以后程序的可伸缩性埋下隐患。即考虑程序的可伸缩性，两者数据量要大致在一个数量级上。

简而言之，Solr 设计多 Core 主要是为了解决生产环境下的如下关键需求：

- ☐ 重建索引；
- ☐ 配置变更影响最小化；
- ☐ 索引合并和分裂；
- ☐ Core 热交换。

分成多个 Core 后，可以单独重建某一个 Core 的索引数据，索引重建效率提高了。由于每个 Core 的配置都是相互独立互不影响的，所以修改某一个 Core 不会对其他线上的 Core 造成影响。而如果是单 Core 的话，多个 Core 之间的索引可以根据业务需求随意合并和分裂，使得索引数据的管理变得更加简便，可控性更强了。最让人眼前一亮的就是 Core 支持热切换，这使得新 Core 上线工作变得简单了。

## 2.1.2 Core 的基本管理

自从 Solr 3.x 开始，Solr Core 就可以通过 Solr 的 Web UI 进行动态管理，你可以动态地添加 Core、卸载 Core、重新加载 Core、Core 热交换等。通过上一节，你已经知道了什么是 Solr Core，那接下来让我们开始动手创建一个 Core。

在创建 Core 之前，我们需要了解 Core 有两个必要的组成元素，即 Schema.xml 和 solrconfig.xml。然后由这两个配置文件衍生出两者依赖的其他配置文件。Core 的目录结构也有要求，Schema.xml 和 solrconfig.xml 必须放置在 solr\_home\core\_name\conf 目录下。solr\_home 即你的 Solr Core 宿主目录，所有 Core 都存放在 solr\_home 目录下，solr\_home 可以通过 web.xml 的 <env-entry> 元素指定，如下：

```
<env-entry>
<env-entry-name>solr/home</env-entry-name>
```

```
<env-entry-value>C:/solr_home</env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

比如这里我将 solr\_home 根目录设置为 C:/solr\_home，然后你需要创建一个 Core 目录，Core 目录名称最好与你的 Core 名称保持一致。然后在 Core 目录下面再创建一个 conf 目录，你的 Core 配置文件将全部存放在这个 conf 目录下。然后可以从 solr 压缩包的这个目录路径（E:\solr-5.3.1\example\example-DIH\solr\solr\conf）下查找相关配置文件作为模板，配置好的效果如图 2-1 所示。

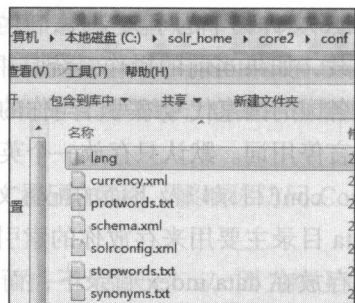


图 2-1 core 的 conf 配置文件示意图

其中 currency.xml 配置文件主要是各种货币之间的汇率常量配置，之所以需要这个配置文件，是因为官方 demo 里提供的 Schema.xml 中定义 currencyField 即货币域，其实如果用不到这个域，你可以在 Schema.xml 中将其注释或者删掉，这样你的 Core 就不必依赖 currency.xml 配置文件了！protwords.txt 可以用来配置那些不需要进行词形还原的英文单词，比如 taken、takes 还原成 take。如果你不想它们被还原，请将其添加至 protwords.txt 字典文件中，一个单词单独占一行。protwords.txt 字典文件是由 KeywordMarkerFilterFactory 这个分词过滤器工厂类加载的，如图 2-2 所示。

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt" ignoreCase="true"
    expand="false"/>
    -->
    <!-- Case insensitive stop word removal.
    -->
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"
    />
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
    <!-- Optionally you may want to use this less aggressive stemmer instead of PorterStemFilterFactory:
    <filter class="solr.EnglishMinimalStemFilterFactory"/>
```

图 2-2 KeywordMarkerFilterFactory 分词过滤器

如果你确信不需要 text\_en 这个域类型，那么可以将 text\_en 这个域类型整个定义部分注释，这样 protwords.txt 这个字典文件也不是必须依赖的。如果你是 Solr 新手，建议保持原状即可。

接下来的 schema.xml 就是重点了，这个配置文件主要用来定义你索引数据需要的域和域类型，即你需要在这里声明索引数据中需要哪些域以及每个域的类型（域类型决定了被该类型修饰的域的域值该如何被索引、如何被分词、如何被存储等）。注意，这里不需要像在 Lucene 里那样去手动新建各种域，然后一个个添加到 Document 对象中，因为创建域以



及添加域这部分工作 Solr 会自动完成。solrconfig.xml 是 Solr Core 必需的一个配置文件，主要用来配置索引创建、查询、Solr 缓存以及 Solr 组件处理器等信息。以 stopwords.txt 为自定义的停用词字典文件主要由停用词过滤器工厂类 StopFilterFactory 加载。synonyms.txt 即自定义的同义词字典文件，它主要由同义词过滤器工厂类 SynonymFilterFactory 加载。至于 lang 文件夹下的字典文件，其实也是停用词字典文件，只不过这里是分各国语言进行动态加载，如果当前的操作系统是中文语言包，那就会加载 stopwords\_zh.txt 字典文件，en、zh 为各国的国家代号缩写。当你的搜索程序需要适配各国语言时，可能会需要配置这里的各国语言停用词。默认只存放一个英文的停用词字典文件模板。

conf 目录以及下面的配置文件准备好了，你还可以在 Core 目录下创建 data 目录，这个 data 目录主要用来存放你的索引数据和 Solr 日志文件，你当前 Core 相关的索引数据都将全部存放在 data\index 目录下，而 Solr 日志文件将全部存放在 tlog 目录下，index 和 tlog 目录不需要手动创建，Solr 会自动判断是否存在该目录，若不存在会自动新建。在创建 Core 阶段，data 目录不是必须手动创建的，即 data 目录在你 Core 添加成功后会自动创建。

这一切准备工作做好之后，你就可以访问 Solr 的 Web 后台，在 Core Admin 模块中，单击 Add Core 按钮来创建 Core，如图 2-3 所示。

其中：

- ❑ name：即你的 Core 名称，注意 Core 名称必须全局唯一，不可重复；
- ❑ instanceDir：表示你的 Core 根目录，这里是相对你的 solr\_home；
- ❑ dataDir：表示你当前 Core 的数据目录，这里是相对上面的 instanceDir；
- ❑ config：表示你的 solrconfig.xml 存放路径，默认是相对你 instanceDir 下的 conf 目录；
- ❑ schema：表示你的 schema.xml 存放路径，默认也是相对你 instanceDir 下的 conf 目录。

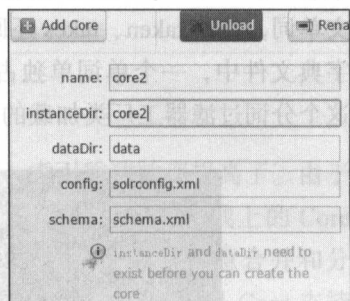


图 2-3 Core Admin 界面

如果创建 Core 的过程中抛出了异常，请仔细阅读异常信息，可以单击 Solr Web 后台左侧的 logging 菜单来查看 Solr 日志，也可以直接进入 data\tlog 目录打开日志文件直接查看日志信息，通过查看日志信息一般都能找到问题所在。

当你的 Core 创建成功后，Solr 会自动在当前 Core 的根目录下生成一个 core.properties 配置文件，这里主要是记录你在添加 Core 时填写的基本信息。Core 创建成功后即表示这个 Core 已经注册到 Solr 中了。Solr 服务可以接受各种请求对这个 Core 进行管理操作，比如当你修改了 Core 的 solrconfig.xml 或 schema.xml，而你又不希望重启你的 servlet 容器（如 tomcat,jetty），这时需要重新加载 Core 使修改后的配置能够生效。注意，由于 Core 加载需要一个过程，在 Core 重新加载尚未完成之前，Solr 请求还是会按照修改之前的配置进行处



理，只有等到 Core 的重新加载操作完全完成之后，后续的 Solr 请求才会按照最新修改的配置进行处理，而旧的 Core 将会被卸载。在 Solr Web UI 的最左侧的 Core Admin 模块中，有个 reload 按钮，请首先选中你需要重新加载的 Core，然后单击该按钮触发 Core 的重新加载操作。reload 操作就相当于先按照新修改的配置再配置一个同名的 Core，等新 Core 创建好再把旧 Core 卸载。那什么又是 Core 的卸载呢？所谓 Core 的卸载其实就是将 Core 从 Solr 中取消注册，这样 Core 就脱离了 Solr 管理，该 Core 目录下的所有文件就与 Solr 没有任何关系了，你可以任意删除他们。卸载操作只是取消 Core 与 Solr 的管理关系，它并不会删除 Core 根目录下的文件。

你可以用 rename 修改一个 Core 的名称，但它只是修改 Core 的名称，影响你访问 Core 的 url，并不会修改该 Core 在硬盘上的 Core 根目录名称。

可能你会有这样的需求：用一个候补的 Core 去替换线上的一个活跃 Core，因为线上的 core 正在运行中，不可能停掉，这时候 swap 热替换就派上用场了。swap 热交换实际就是 Core 目录交换。举个例子，假如你有两个 Core (core1 和 core2)，你用 core2 去交换 core1，原本 core1 对应的 Core 目录为 core1，core2 对应的 Core 目录为 core2，swap 操作之后，core1 对象的 Core 目录变成 core2 交换前对应的 Core 目录 core2，core2 的 Core 目录变成原 core1 的 Core 目录。swap 热交换并不影响原运行中的 Core，被替换掉的旧 Core 会一直保持运行状态，这样做是为了便于你随时再 swap 替换回去做撤销。

当你选中一个 Core，在界面右侧会显示当前 Core 的一些状态信息，如 instanceDir、DataDir、numDocs 等。这其实调用的是 STATUS 接口，此接口的 URL 地址为：

```
http://localhost:8080/solr/admin/cores?action=STATUS&core=core0
```

其中，Core 参数表示你要查看哪个 Core 的状态信息，core0 代表的是某个 Core 的名称。该 Core 参数并不是必须的，如果你不指定 Core 参数，比如这样：

```
http://localhost:8080/solr/admin/cores?action=STATUS
```

即表示你需要查看所有 Core 的状态信息。

当然多个 Core 的操作远不止 Solr Web 后台提供的那几个，有时候你可能想要将多个索引目录里的数据合并到一个新的索引目录中，即 Merge index。关于索引合并，Solr 提供的接口 URL 为：

```
http://localhost:8080/solr/admin/cores?action=mergeindexes&core=core0&indexDir=/opt/solr/core1/data/index&indexDir=/opt/solr/core2/data/index
```

参数 mergeindexes 表示执行的是索引合并操作，Core 参数表示索引合并到哪个 Core，两个 indexDir 参数表示两个索引目录，即待合并的索引目录，可以有多个 indexDir。注意，这里的 indexDir 可以与 Solr Core 无关，它可以是通过 Lucene 创建的索引目录，只要该索引目录里的索引数据结构兼容当前 Solr 版本即可。另外 core0 必须是一个在 Solr 中已经存在的 Core。indexDir 是针对 Lucene 索引合并而设计，自然有针对 Solr 的索引合并。Solr 里的

索引数据基本是分 Core 存放的, 一个 Core 对应一个索引目录即 `\dataDir\index`, 因为对于一个 Core 来说, 只要名称确定, 那该 Core 的索引目录也是确定的, 所以只需要指定 Core 名称即可实现索引合并, 这也是索引合并的另一个用法:

```
http://localhost:8080/solr/admin/cores?action=mergeindexes&core=core0&srcCore=core1&srcCore=core2
```

其中 `srcCore` 参数表示待合并的索引目录, 可以执行多个; `Core` 参数表示目标索引目录即合并后的索引数据存放在哪个 Core 下。注意: 这里 `Core` 参数和 `srcCore` 参数指定的 Core 名称指代的 Core 必须在 Solr 存在, 否则索引合并会失败。

索引合并除了可以使用 Solr 提供的接口以外, 还可以使用 Lucene 提供的一个索引工具类 `IndexMergeTool`, 该类在 `org.apache.lucene.misc` 的 `lucene-misc-version.jar` 包中, 而 `lucene-misc-version.jar` 又依赖 `lucene-core-version.jar`, 所以 `IndexMergeTool` 的使用方式如下:

```
java -cp lucene-core-version.jar:lucene-misc-version.jar org.apache.lucene.misc.IndexMergeTool C:/newindex C:/solr_home/core1/data/index C:/solr_home/core2/data/index
```

其中 `version` 代指 Lucene 的版本号, `C:/newindex` 表示合并后的索引目录, 后面两个为待合并的索引目录, 当然可以有 `N` 个待合并的索引目录, 唯一要注意的是合并后的索引目录参数必须放置在所有待合并的索引目录参数的前面。下面是 `IndexMergeTool` 工具类的源码, 其实就简单几行代码:

```
public class IndexMergeTool {
    public static void main(String[] args) throws IOException {
        if (args.length < 3) {
            System.err.println("Usage: IndexMergeTool <mergedIndex><index1><index2>[index3] ...");
            System.exit(1);
        }
        FSDirectory mergedIndex = FSDirectory.open(Paths.get(args[0]));
        IndexWriter writer = new IndexWriter(mergedIndex,
            new IndexWriterConfig(null).setOpenMode(OpenMode.CREATE));
        Directory[] indexes = new Directory[args.length - 1];
        for (int i = 1; i < args.length; i++) {
            indexes[i - 1] = FSDirectory.open(Paths.get(args[i]));
        }
        System.out.println("Merging...");
        writer.addIndexes(indexes);
        System.out.println("Full merge...");
        writer.forceMerge(1);
        writer.close();
        System.out.println("Done.");
    }
}
```

其中的核心代码: `writer.forceMerge(1)`; 强制将索引合并为一个段文件。不过要注意:

`writer.forceMerge(1)`；其实等价于 `writer.forceMerge(1,true)`；第二个参数表示在合并索引过程中是否需要锁住当前 `IndexWriter` 对象，直到索引合并完成，否则使用当前 `IndexWriter` 做的任何操作都将被阻塞。

有索引合并，相对的，自然有索引拆分，即 `Split Index`。`Solr` 已经提供了 `Split` 接口，该接口用于将一个索引分割成 2 个或 2 个以上的索引。它接收以下几个参数：

- `Core`：要对哪个 `Core` 的索引进行分割；
- `path`：分割后的索引写入到哪个索引目录，这是一个多值参数，即它可以指定多次；
- `targetCore`：这个参数表示把源 `Core` 分割成 `N` 个 `targetCore`，这里 `targetCore` 指代一个在 `Solr` 中已经存在的 `Core` 名称。

下面是几个 `Split` 接口的调用示例：

1) `http://localhost:8080/solr/admin/cores?action=SPLIT&core=core0&path=/path/to/index/1&path=/path/to/index/2`;

2) `http://localhost:8080/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1&targetCore=core2`。

### 2.1.3 Core Http 接口

#### 1. STATUS

获取指定 `Core` 的运行状态信息，若没有指定 `Core`，那么就是返回所有 `Core` 的运行状态。

```
http://localhost:8983/solr/admin/cores?action=STATUS&core=core0;
http://localhost:8983/solr/admin/cores?action=STATUS.
```

#### 2. CREATE

创建一个新 `Core`，前提是 `Core` 目录已经提前创建好且 `Core` 依赖的 `solrconfig.xml`/`schema.xml` 都创建并配置正确，如果提供了 `persist=true` 参数，那么新 `Core` 的配置会被保存到 `solr.xml` 中。

```
http://localhost:8983/solr/admin/cores?action=CREATE&name=coreX&instanceDir=path_
to_instance_directory&config=config_file_name.xml&schema=schema_file_name.xml&
dataDir=data
```

`instanceDir` 参数是必需的，`config`、`schema` 以及 `dataDir` 参数是可选的，`Solr 4.3` 版本之后还提供了如下两个可选参数：

- `loadOnstartup` 表示当 `Solr` 启动时是否加载此 `Core`；
- `transient` 表示当瞬态 `Core` 数量达到了 `solr.xml` 中 `<cores>` 元素配置的 `transientCacheSize` 参数值时，是否需要卸载当前 `Core`。

#### 3. RELOAD

重建加载指定 `Core`，在新 `Core` 加载未完成之前，对于该 `Core` 的请求还是由旧 `Core` 来

处理，直到新 Core 加载完成，那么旧 Core 就会被卸载，之后的请求将交由新 Core 来处理。

```
http://localhost:8983/solr/admin/cores?action=RELOAD&core=core0
```

当你修改了 solrconfig.xml 或 schema.xml 后，通过使用 reload 功能来重新加载 Core 可以实现无须停止 Core 或重启 Server 容器即可实现 Core 配置的动态更新。

#### 4. RENAME

重命名一个 Core 的访问名称，下面是一个将 Core 的名称由 core0 重命名为 core5 的示例：

```
http://localhost:8983/solr/admin/cores?action=RENAME&core=core0&other=core5
```

#### 5. SWAP

自动交换两个已知 Core 的名称，当你需要使用一个替补 Core 来替换一个线上活跃中的 Core，并且要求替换后原来的旧 Core 能继续运行以保证你能随时撤销回原先的状态，这个时候，swap 热交换就显得很有用。

```
http://localhost:8983/solr/admin/cores?action=SWAP&core=core1&other=core0
```

#### 6. UNLOAD

将指定 Core 从 Solr 中移除，在 Core 被完全移除之前，之前未处理完的 Core 请求会继续处理，之后发送给该 Core 的新请求就不再受理。

```
http://localhost:8983/solr/admin/cores?action=UNLOAD&core=core0
```

可选参数如下：

☐ deleteIndex：移除 Core 的同时是否需要删除该 Core 下的索引数据，默认不会删除 Core 的索引数据；

```
http://localhost:8983/solr/admin/cores?action=UNLOAD&core=core0&deleteIndex=true
```

☐ deleteDataDir：移除 Core 的同时是否需要删除当前 Core 的 dataDir 目录；

☐ deleteInstanceDir：移除 Core 的同时是否需要删除当前 Core 实例的根目录；

```
http://localhost:8983/solr/admin/cores?action=UNLOAD&core=core0&deleteIndex=true
```

#### 7. LOAD

加载指定 Core

```
http://localhost:8983/solr/admin/cores?action=LOAD&core=core0&persist=true
```

persist 参数表示是否需要将当前 Core 的配置保存到 solr.xml 配置文件中

### 2.1.4 添加索引至 Core

前面我们已经了解到了如何创建一个 Core，接下来我们继续学习如何往 Core 里添加索

引数据。索引数据是全文搜索的前提，如果连搜索对象都没有，就不用谈搜索了。Solr 的 Web 后台已经提供了这样的功能界面，可以方便我们添加索引数据，如图 2-4 所示。



图 2-4 通过 Solr 后台界面添加索引数据至 Core 中

先选择一个 Core，然后单击左侧的 Documents 菜单，打开 Solr 的索引添加界面。下面简单说明添加索引表单页面几个元素的含义：

- Request-Handler：表示添加索引接口对应的请求 URL，只不过这里省去了固定前缀 `http://localhost:8080/solr/{coreName}`；
- Document Type：表示你要添加的 Document 类型，Solr 支持了常见的几种数据格式，如 CSV，JSON，XML 等；
- Commit Within：表示索引必须在限定的时间内完成提交，否则放弃操作，单位为毫秒，这个设置主要是为了防止索引提交长期阻塞；
- Overwrite：覆盖的意思，即是否需要根据索引数据的 uniqueKey 来覆盖之前添加的 Document，只要 uniqueKey 值 equals，那么就会进行 Document 覆盖。此参数默认值为 true。注意：前提是你有在 schema.xml 里配置 uniqueKey，否则即使你设置 Overwrite 为 true，也不会覆盖；
- Boost：此参数用来设置当前你要添加的索引的权重值，默认值为 1.0；
- Documents：这里就是你要填写的索引数据，它根据 Document Type 值不同，会有不同的编写格式。

了解了这些参数含义，有助于你更好地去使用 Solr 为我们提供的这个索引添加功能。当你选择一个 Document Type 时，Documents 文本框里会有相应的示例提示，这应该很好理解，id 即你的主键域，然后“域名称”：“域值”，如果域值是 bool 值或者数字，域值两

头可以不用添加双引号。不过需要注意的是,在添加索引之前,你的域需要提前在 schema.xml 中定义好。

比如 JSON 格式:

```
{"id":1,"title": "change me"}
```

这里一个花括号表示一个 Document 对象,里面的 id、title 就是该 document 包含的域,域名称冒号后面紧跟着的就是域值。简单的域值直接用双引号包裹起来或者用数字表示,特殊情况下,比如该域为多值域时,即 multiValued=true 时,域值需要使用 [] 中括号包裹起来。比如这样:

```
{"id":1,"title": ["change me 1","change me 2","change me 3"]}
```

Document 包含的域需要提前在 schema.xml 中定义,比如这样:

```
<field name="title" type="string" indexed="true" stored="true" required="false" />
```

同理还有 XML、CSV 格式,Documents 文本框里都对应有相关的示例提示,按照示例提示去添加这应该不难,比如 CSV 的 document 填写格式是这样的:

```
id,title
change.me,change.me
```

第一行是 document 包含的域,多个域名称之间用逗号分隔,第二行是该域对应的域值,同理也是逗号分隔。至于 schema.xml 配置的详细介绍留给后续再做讲解,这里暂不赘述。

## 2.2 Solr DIH

大多数的应用程序将数据存储于关系数据库、xml 文件中。对这样的数据进行搜索是很常见的应用。DIH (Data Import Handler) 提供了一种可配置的方式向 Solr 中导入数据,可以一次性全量导入,也可以增量导入。

### 2.2.1 索引文件夹下的文本文件

通过 Solr 后台的 Documents 添加界面去添加索引,由于只能单个 Document 添加,效率太低。可能你的索引数据全都保存在文本文件里,比如 txt 文件,那么,如何批量去索引某个文件夹下的文本文件呢? Solr 的 Web 后台其实提供了数据导入功能简称 DIH,如图 2-5 所示。

开始之前,你需要了解 DIH 能帮你做些什么:

- 1) 它能读取数据库的数据并创建索引;
- 2) 它能够基于配置的方式把数据里表的列甚至多个表的数据聚合并解析成一个 Document;



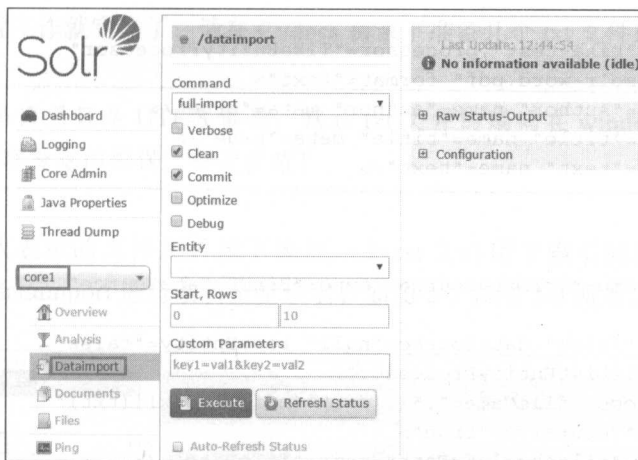


图 2-5 Solr 后台的 dataImport 功能界面

- 3) 它支持基于配置的全量和增量数据导入;
- 4) 它能实现基于配置的定时全量和增量索引;
- 5) 它能基于 HTTP 方式读取并索引 XML 文件;
- 6) 它支持各种基于插件式的 datasource 和 formate 配置。

要开启使用 DIH 功能, 你需要在 solrconfig.xml 配置文件中配置 dataimport 请求处理器, 并指定 data-config.xml 配置文件加载路径:

```
<requestHandler name="/dataimport" class="solr.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

上面的 config 参数用于指定 data-config.xml 的加载路径, 默认是相对你当前 Core 的 conf 目录, 当然你也可以指定为绝对路径, 如: C:/xxxx/xxxx/data-config.xml。当然如果你还需要配置增量导入, 那么你还需要再配置一个 DataImportHandler, 此时 name="deltaimport", 然后增量导入也需要对应一个 delta-data-config.xml。你需要指定依赖的 jar 包加载路径:

```
<dataDir>C:\solr_home\core1\data</dataDir>
<lib dir="./lib" regex=".*\.jar"/>
```

依赖的 jar 包如图 2-6 所示。

然后重点是配置我们的 data-config.xml 了, 配置内容如下:

```
<dataConfig>
  <dataSource name="fileDataSource" type="File-
DataSource" />
<!--
```

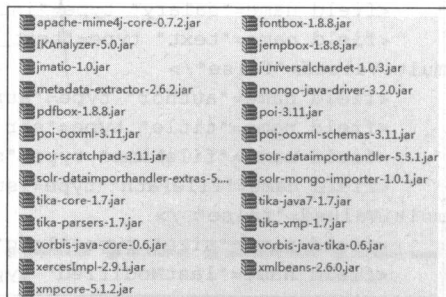


图 2-6 core 依赖 Jar 包

```

<document>
<entity name="tika-test" processor="TikaEntityProcessor"
url="C:/docs/solr-word.pdf" format="text">
<field column="Author" name="author" meta="true"/>
<field column="title" name="title" meta="true"/>
<field column="text" name="text"/>
</entity>
</document>

-->
<dataSource name="urlDataSource" type="BinURLDataSource" />
<document>
<entity name="files" dataSource="null" rootEntity="false"
processor="FileListEntityProcessor"
baseDir="c:/docs" fileName=".*\\. (doc) | (pdf) | (docx) | (txt)"
onError="skip" recursive="true">
<field column="fileAbsolutePath" name="filePath" />
<field column="fileSize" name="size" />
<field column="fileLastModified" name="lastModified" />
<entity processor="PlainTextEntityProcessor" name="txtfile" url="${files.file-
AbsolutePath}" dataSource="fileDataSource">
<field column="plainText" name="text"/>
</entity>
</entity>
</document>
</dataConfig>

```

`baseDir` 表示获取这个文件夹下的文件, `fileName` 支持使用正则表达式来过滤一些 `baseDir` 文件夹下你不想被索引的文件, `processor` 是用来生成 Entity 的处理器, 而不同 Entity 默认会生成不同的 Field 域。FileListEntityProcessor 处理器会根据指定的文件夹生成多个 Entity, 且生成的 Entity 会包含 `fileAbsolutePath`, `fileSize`, `fileLastModified`, `fileName` 这几个域。注意, 这几个域是 FileListEntityProcessor 内置的, 不能随意更改。recursive 表示是否递归查找子目录下的文件, onError 表示当出现异常时是否跳过这个条件不处理。

然后我们需要在 `schema.xml` 中定义域:

```

<field name="userName" type="string" indexed="true" stored="true" omitNorms="true"/>
<field name="sex" type="boolean" indexed="true" stored="true" omitNorms="true"/>
<field name="birth" type="cndate" indexed="true" stored="true" omitNorms="true"/>
<field name="salary" type="int" indexed="true" stored="true" omitNorms="true"/>
<field name="text" type="text_ik" indexed="true" stored="true" omitNorms="true"
multiValued="false"/>
<field name="author" type="string" indexed="true" stored="true" />
<field name="title" type="string" indexed="true" stored="true" />
<field name="fileName" type="string" indexed="true" stored="true" />
<field name="filePath" type="string" indexed="true" stored="true" required="true"
multiValued="false" />
<field name="size" type="long" indexed="true" stored="true" />
<field name="lastModified" type="cndate" indexed="true" stored="true" />
<field name="id" type="string" indexed="true" stored="true" required="false" multi-
Valued="false" />

```

到此，Solr 配置工作就完毕了，请在 C:/docs 目录下准备几个 txt 文件用于索引创建测试。

**注意** txt 文件编码请保证是 UTF-8 编码，默认 txt 文件的编码在 windows 下是 GBK。这是很多小白容易犯的错误，特此提醒！

在 baseDir 参数表示的文件夹目录下添加一个 txt 文件用于索引测试，然后重启你的 Tomcat，选择 full-dataimport 进行索引全量索引，如图 2-7 和图 2-8 所示。

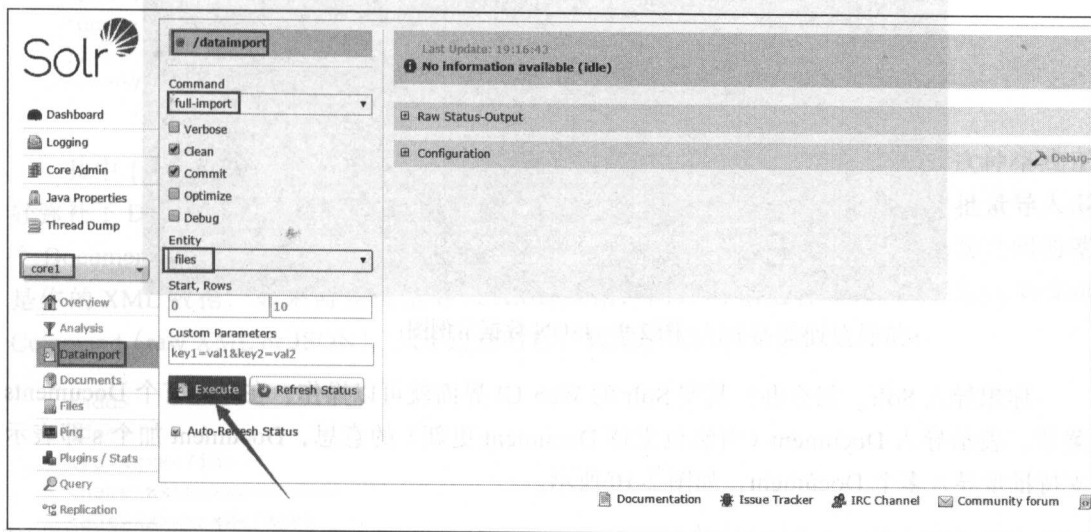


图 2-7 dataImport 操作演示图

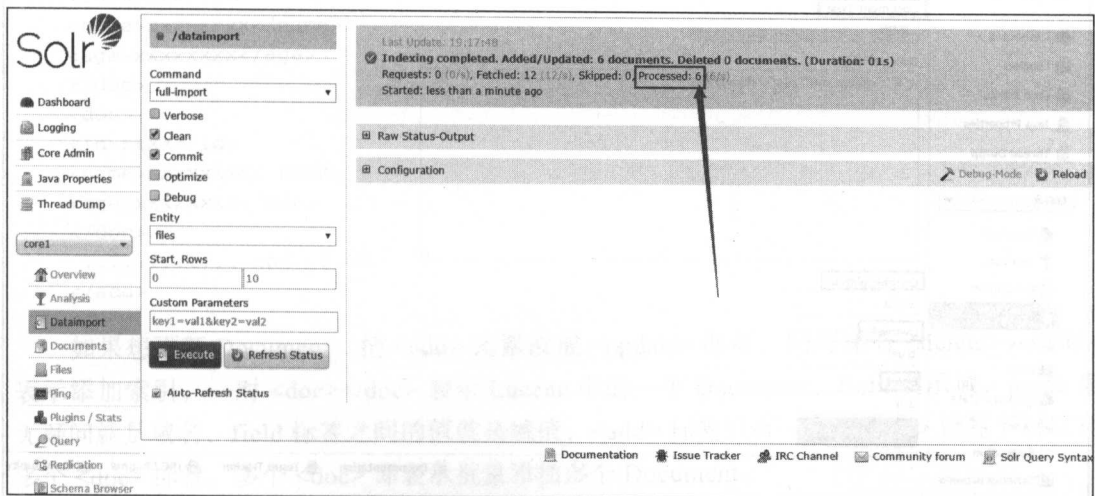


图 2-8 dataImport 操作成功效果图

图 2-8 中 Request 表示 Solr 接收到了多少个 http 请求, Fetched 表示 Solr 提取到了多少个文件, Skipped 表示 Solr 跳过了多少个文件未处理, Processed 表示 Solr 成功导入了多少个 Document。

## 2.2.2 索引 JSON/XML/CSV 文件

假定你有这样一堆 JSON 数据, 如图 2-9 所示。

```

1 2
3 {
4   "id": "1", "name": "Red Lobster", "city": "San Francisco, CA", "type": "Sit-down Chain", "state": "California", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
5   "id": "2", "name": "Red Lobster", "city": "Atlanta, GA", "type": "Sit-down Chain", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
6   "id": "3", "name": "Red Lobster", "city": "New York, NY", "type": "Sit-down Chain", "state": "New York", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
7   "id": "4", "name": "McDonalds", "city": "San Francisco, CA", "type": "Fast Food", "state": "California", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
8   "id": "5", "name": "McDonalds", "city": "Atlanta, GA", "type": "Fast Food", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
9   "id": "6", "name": "McDonalds", "city": "New York, NY", "type": "Fast Food", "state": "New York", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
10  "id": "7", "name": "McDonalds", "city": "Chicago, IL", "type": "Fast Food", "state": "Illinois", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
11  "id": "8", "name": "McDonalds", "city": "Austin, TX", "type": "Fast Food", "state": "Texas", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
12  "id": "9", "name": "Pizza Hut", "city": "Atlanta, GA", "type": "Sit-down Chain", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
13  "id": "10", "name": "Pizza Hut", "city": "New York, NY", "type": "Sit-down Chain", "state": "New York", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
14  "id": "11", "name": "Pizza Hut", "city": "Austin, TX", "type": "Sit-down Chain", "state": "Texas", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
15  "id": "12", "name": "Freddy's Pizza Shop", "city": "Los Angeles, CA", "type": "Local Sit-down", "state": "California", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
16  "id": "13", "name": "The Iberian Pig", "city": "Atlanta, GA", "type": "Upscale", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
17  "id": "14", "name": "Sprig", "city": "Atlanta, GA", "type": "Local Sit-down", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
18  "id": "15", "name": "Starbucks", "city": "San Francisco, CA", "type": "Coffee Shop", "state": "California", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
19  "id": "16", "name": "Starbucks", "city": "Atlanta, GA", "type": "Coffee Shop", "state": "Georgia", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
20  "id": "17", "name": "Starbucks", "city": "New York, NY", "type": "Coffee Shop", "state": "New York", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
21  "id": "18", "name": "Starbucks", "city": "Chicago, IL", "type": "Coffee Shop", "state": "Illinois", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
22  "id": "19", "name": "Starbucks", "city": "Austin, TX", "type": "Coffee Shop", "state": "Texas", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
23  "id": "20", "name": "Starbucks", "city": "Greenville, SC", "type": "Coffee Shop", "state": "South Carolina", "tags": ["Fast Food", "Sit-down Chain", "Chain"], "tags": ["Fast Food", "Sit-down Chain", "Chain"]
24 }

```

图 2-9 JSON 数据示例图

你想导入 Solr, 怎么办? 其实 Solr 的 Web UI 界面就可以操作, 在左侧有个 Documents 菜单, 表示导入 Document (当然也支持 Document 更新) 的意思, Document 加个 s 即表示支持批量导入多个 Document, 如图 2-10 所示。

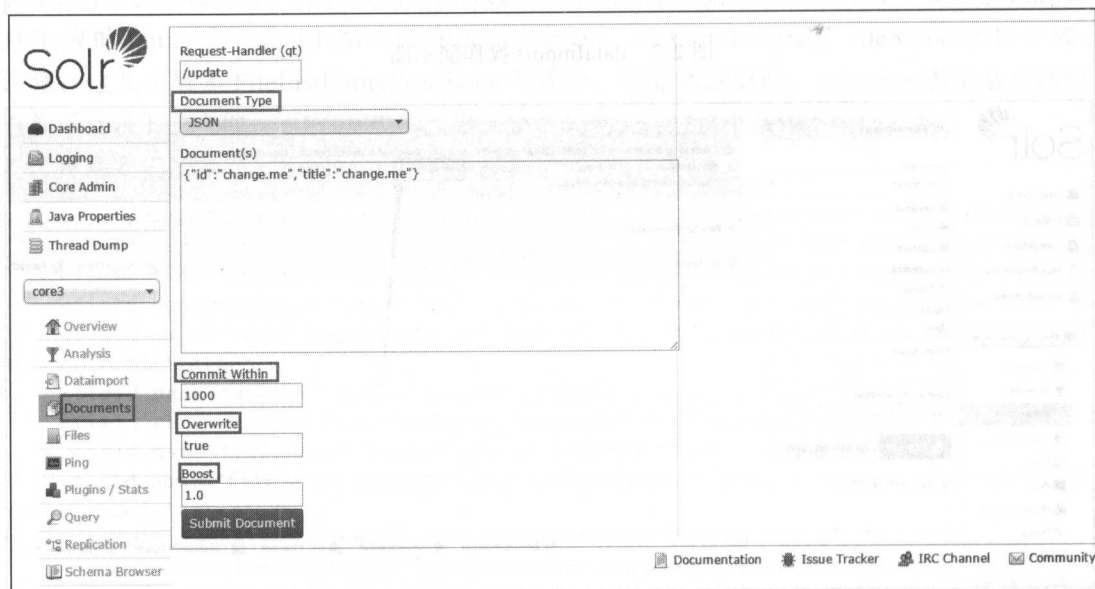


图 2-10 Solr Document Update 操作界面

如果只有单个 JSON 对象需要导入，那直接选择 Document Type 为 JSON 即可，当你选择 Document Type 为 JSON 后，Document(s) 文本框会提示一个示例。如果你需要批量导入多个 JSON 对象，这里请选择 Document Type 为 Solr Command (raw XML or JSON)，只不过这时候 JSON 数据格式就有特殊要求了，你的 JSON 数据格式需要这样定义：

```
{
  "add":{
    "doc": {.....}
  },
  "add":{
    "doc": {.....}
  },
  .....// 省略
}
```

其中 {.....} 部分就是你的 Document 对象，其余部分为固定格式。使用这种格式正好弥补了 Document Type 为 JSON 这种方式只能一条一条导入的缺点，当需要批量导入多个 Document 时，采用这种格式支持批量导入多个 Document。<doc></doc> 标签之间的就是你的 XML 数据，如果需要批量导入 XML 数据，同样可以选择 Document Type 为 Solr Command (raw XML or JSON)，只不过这时候，数据格式应该是类似这样的：

```
<add>
<doc>
<id>xxxx</id>
<name>xxxxxxxx</name>
<age>xxxxxxxx</age>
</doc>
<doc>
<id>xxxx</id>
<name>xxxxxxxx</name>
<age>xxxxxxxx</age>
</doc>
<doc>
<id>xxxx</id>
<name>xxxxxxxx</name>
<age>xxxxxxxx</age>
</doc>
..... and so on
</add>
```

如果想更新 Document，把 <add> 元素改成 <update> 即可，同理还有 <delete>，<add> 表示添加索引，一对 <doc></doc> 表示 Lucene 中的一个 Document，field 表示域，name 毫无疑问就是域名，field 标签之间的值就是域值，<add> 标签只有一个，<add> 标签下可以有多个 <doc> 标签，多个 <doc> 即表示批量添加多个 Document。

<add> 标签还有 2 个可选属性，

1) `overwrite: "true" | "false"`, 默认为 `false`, 表示对于拥有相同 `uniqueKey` 的 Document 是否需要覆盖, `uniqueKey` 表示 document 的唯一主键, 类似数据库表的主键;

2) `commitWithin`: 表示 Document 必须在指定的毫秒数内提交成功, 否则就放弃提交。还可以为某个 Document 设置权重, 比如:

```
<add>
<doc boost="2.5">
<field name="employeeId">05991</field>
<field name="office" boost="2.0">Bridgewater</field>
</doc>
</add>
```

如何添加多值域?

```
<add>
<doc>
<field name="employeeId">05991</field>
<field name="skills" update="set">Python</field>
<field name="skills" update="set">Java</field>
<field name="skills" update="set">Jython</field>
</doc>
</add>
```

如何将某个域的值设为 `null`?

```
<add>
<doc>
<field name="employeeId">05991</field>
<field name="skills" update="set" null="true" />
</doc>
</add>
```

还可以在 `<add>` 标签下添加:

```
<commit/>
<optimize/>
```

类似于在 Lucene 里显式地调用:

```
writer.commit();
writer.optimize();
```

如何根据 ID 删除 Document? (注意这里说的 id 指的是 `uniqueKey` 指定的域, `uniqueKey` 是在 `schema.xml` 中定义的, 不要与 Document 的文档 ID 混为一谈)

```
<delete><id>05991</id></delete>
```

如何根据一个 Query 删除一个 Document 呢?

```
<delete>
```



```
<query>office:Bridgewater</query>
</delete>
```

office 表示域名, bridgewater 表示域值, 默认创建的是 TermQuery, 域值可以有通配符, 可以是正则表达式, 可以使用 QueryParser 表达式表示。

你只需要在 Documents 这个多行文本框里按照上述的数据格式要求去填写即可。如果你的 JSON 数据不是上述要求的格式, 你需要按照要求做个转换, 也许你的数据格式已经规范好了, 无法修改数据格式, 或者你有很多这样的 JSON 数据存在于一个文件中, 导致数据格式转换工作量很大, Solr 后台的 Documents 数据导入功能无法满足你的需求, 这时候你需要自定义 EntityProcessor 进行功能扩展, 这个留到后面再做讲解。

## 2.2.3 使用 Tika 索引 Word/Excel/PDF

有时候我们需要索引的数据可能存在于 PDF、Word、Excel 等文件中, 我们需要去解析文件从中获取数据然后建立索引, 而在 Solr 中这类文件解析工作由 Tika 负责。Tika 将诸如 PDF、Word、Excel 的文件统一抽象为富文本文件, 然后定义一套接口去提取它们的内容。使用者可直接调用该接口, 而不用考虑文件类型以及不同类型文件的提取过程。这里不准备详细讲解 Tika 如何使用, 感兴趣的读者可自行上网查阅。下面开始正式讲解 Solr 中如何索引 PDF 文件。

首先你需要添加 Tika 相关 jar 包, 而 Tika 解析 PDF 文件又是借助第三方的 POI 和 PDF-Box。这些 jar 包在 Solr 的 zip 压缩包都能找到, 具体路径为:

E:\solr-5.3.1\contrib\extraction\lib, 需要导入的 jar 包如图 2-11 所示。

首先你需要在 Core 根目录下新建一个 lib 目录来存放图 2-11 中显示依赖的 jar 包, 如图 2-12 所示。

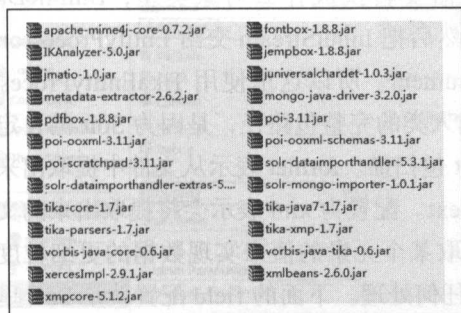


图 2-11 启用 Tika 需要的 jar 包

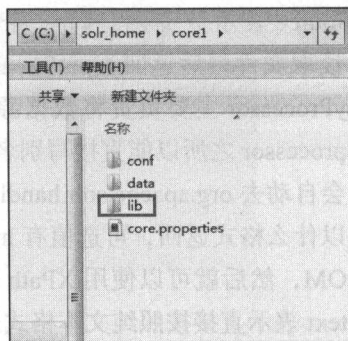


图 2-12 Core 目录中创建 lib 目录

然后我们需要在 solrconfig.xml 配置文件里配置外部 jar 包的加载路径, 如图 2-13 所示。

dir 参数里的 “/” 即表示当前 Core 根目录，regex 表示一个正则表达式，目的就是为了批量指定加载哪些 jar 包，当然也可以直接将依赖的 jar 包放到 Tomcat webapps 目录下部署的 Solr 程序的 WEB-INF\lib 目录下。那么有的同学可能就要发问了：把 jar 包放在 Core 的 lib 目录和直接放到 Tomcat 里两者有什么区别？首先，jar 包的加载时机不同，你把 jar 包放在 Tomcat 下，jar 包在 Tomcat 启动时就立即加载了；而放在 Core 的 lib 目录，是在 Solr 初始化完成该 Core 后根据 solrconfig.xml 里 <lib> 元素配置的规则去加载 jar。再者，放到 Core 的 lib 目录下可以根据每个 Core 的需求去加载 jar，每个 Core 依赖的 jar 分开管理更方便维护；而统一放到 Tomcat 下则杂乱无章，当需要增删 jar 包时，你很难弄清楚可能会影响哪几个 Core。

对于 jar 包路径放置问题，我的建议是，对于每个 Core 都共用的 jar 建议放到 Tomcat 下，而对于每个 Core 单独依赖的 jar 可以放置在当前 Core 的 lib 目录下。

然后在当前 Core 的 solrconfig.xml 中，配置 dataimport 处理器并指定 dataconfig.xml 配置文件的加载路径，如图 2-14 所示。

在当前 Core 的 solrconfig.xml 的同级目录下新建一个 data-config.xml 配置文件，如图 2-15 所示，对其进行相关配置：

dataSource 表示数据源，即你要索引的文件需要转换成什么对象类型，BinFileDataSource 会读取文件最后返回 InputStream 输出流，然后把 InputStream 交给 EntityProcessor 处理，EntityProcessor 主要负责将数据源转换成 Document。所以这里使用 TikaEntityProcessor 处理器，processor 之所以能直接写别名而不需要输入类的完整包路径，是因为 Solr 的自定义类加载器会自动去 org.apache.solr.handler.dataimport 包扫描。format 表示从文件中提取出来的文本需要以什么格式返回，可选值有 html、xml、text。配置为 xml 表示会将提取出来的文本映射成 DOM，然后就可以使用 XPath 表达式去获取某个元素的值来实现数据的更细粒度的提取了。text 表示直接按照纯文本格式返回，不做任何处理。下面的 field 配置就是 Solr 里的域与 Tika 提取的元数据项的一个映射。Tika 内部会对富文本文件解析并提取 Author、Title、Language、Publishers 等元数据项，而这些元数据项的名称是在 Metadata.java 类中定义的，该类包含在 tika-core-version.jar 中，具体需要你们去观摩源码。富文本文件提取出来的内容映射到哪个域呢？请看图 2-16。

```

43  used to specify an alternate directory to load all index data
44  other than the default ./data under the Solr home. If
45  replication is in use, this should match the replication
46  configuration.
47  -->
48  <dataDir>{solr.data.dir}</dataDir>
49  -->
50  <dataDir>C:\solr-home\core1\data</dataDir>
51  <lib dir="/lib" regex=".*\.jar" />
52  <!-- The DirectoryFactory to use for indexes.
53  -->
54  solr.StandardDirectoryFactory is filesystem
55  based and tries to pick the best implementation for the current
56  JVM and platform. solr.NRTDirectoryFactory, the default
57  uses solr.StandardDirectoryFactory and copies small files in

```

图 2-13 配置 Core 依赖的 jar 包加载路径

```

-->
<requestHandler name="/dataimport" class="solr.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>

```

图 2-14 配置 data-config.xml

```

<dataConfig>
  <dataSource name="fileDataSource" type="BinFileDataSource" />
  <document>
    <entity name="tika-test" processor="TikaEntityProcessor"
      url="C:/docs/solr-word.pdf" format="text">
      <field column="Author" name="author" meta="true"/>
      <field column="title" name="title" meta="true"/>
    </entity>
  </document>
</dataConfig>

```

图 2-15 data-config 配置示例

```

162 IOUtils.closeQuietly(is);
163 for (Map<String, String> field : context.getAllEntityFields()) {
164     if (!"true".equals(field.get("meta"))) continue;
165     String col = field.get(COLUMN);
166     String s = metadata.get(col);
167     if (s != null) row.put(col, s);
168 }
169 if(!"none".equals(format) ) row.put("text", sw.toString());
170 done = true;
171 return row;
172 }

```

图 2-16 TikaEntityProcessor 类部分源码

没错，Solr 把 Tika 提取出来的文本内容放到 text 域里。意思就是说从 PDF 提取出来的文本内容默认是存放在 text 域。那么你可能要问了，可不可以把文本内容存放在一个自定义的域上？答案是可以的。你需要设置 `fmap.content=description` 参数，即把 Tika 提取出来的 content 映射到 description 域，前提是你已经在 schema.xml 里已经声明了该域。不过这个参数在 Solr Web 后台无法设置，具体内容后续讲解到 solrJ 时再详述。那如果你想再增加几个域呢？比如你在做部门的文件搜索系统，可能会需要添加一个部门域来表示文件属于哪个部门的，而 Tika 不可能知道跟你业务相关的部门信息。再比如你可能需要为每个文件添加一个唯一主键来标识每个文件的唯一性，这时，可以为你的自定义域名称加上 literal. 前缀。对于 TikaEntityProcessor 处理器而言，凡是以 literal. 开头的域都表示自定义域，即该域的域值需要用户自己指定，Solr 不会为该域赋值。

然后我们需要在 C:/docs 目录下放一个 PDF 文件进行测试，solr-word.pdf 这个测试 PDF 文件在 solr 5 的解压目录下可以找到，文件路径为 E:/solr-5.3.1/example/exampledocs。到此准备工作就完成了，重启 Tomcat，访问你的 Solr Web UI 进行测试，如图 2-17 所示。

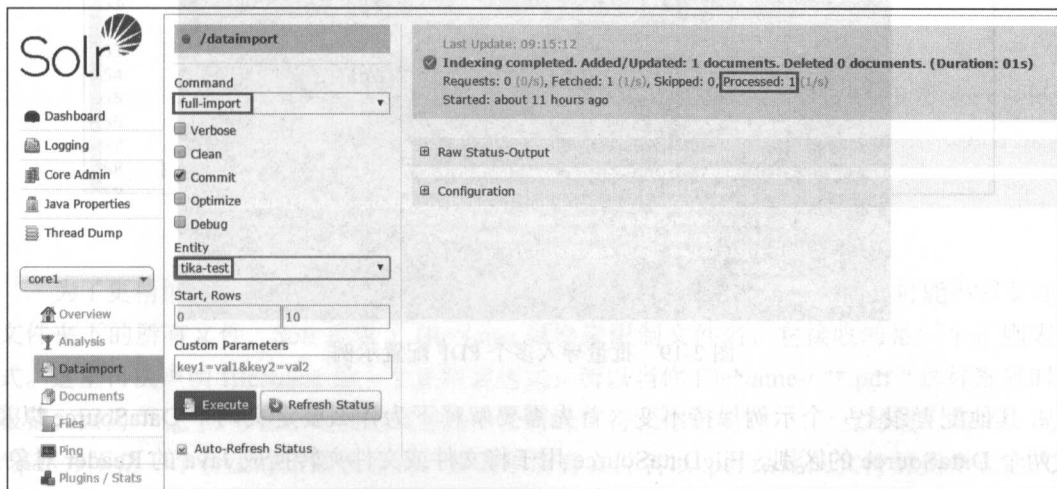


图 2-17 使用 Tika 导入数据

如果你执行后看到如图 2-17 所示的效果，就表明 PDF 导入 Solr 成功了。为了验证 PDF 是否成功导入 Solr，可以切换到 Query 菜单进行查询验证，如图 2-18 所示。

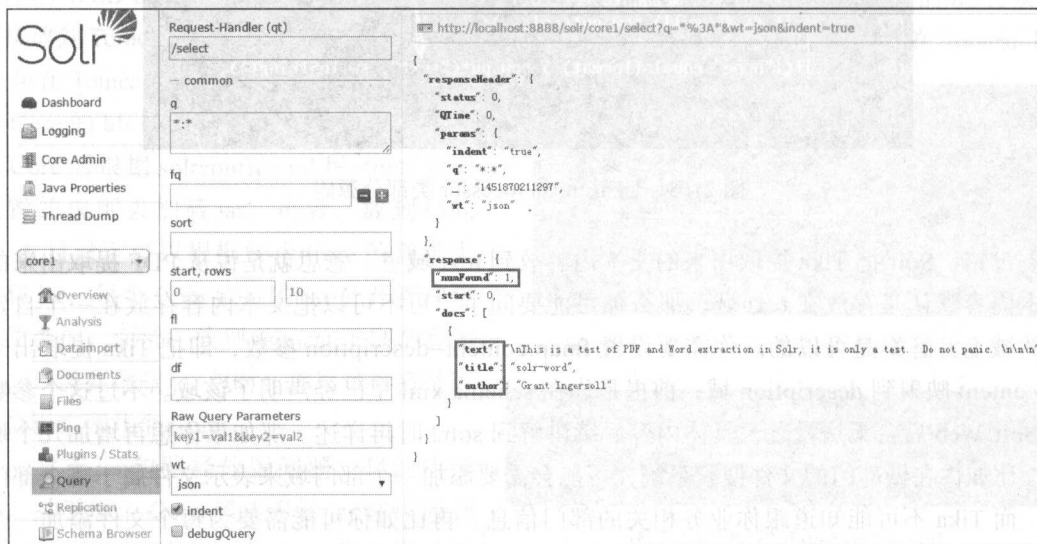


图 2-18 Tika 导入数据成功效果图

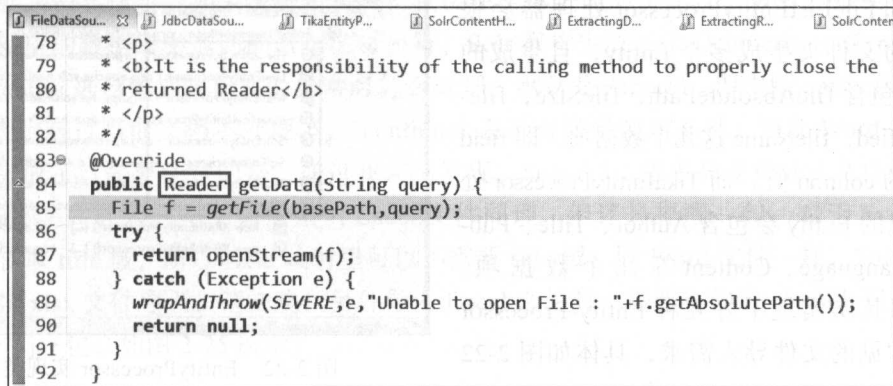
到这儿，单个 PDF 文件索引就完成了。但现实中，我们可能不单单只是索引一个 PDF 文件，我们可能需要索引  $N$  个 PDF 文件，此时该如何配置呢？别担心，Solr 已经帮你考虑到了，它设计了一个 `FileListEntityProcessor` 处理器类，专门用于批量处理指定文件夹下的文件。`data-config.xml` 配置示例如图 2-19 所示。

```
<dataConfig>
  <dataSource name="fileDataSource" type="FileDataSource" />
  <dataSource name="binfileDataSource" type="BinfileDataSource" />
  <document>
    <entity name="pdfs" dataSource="fileDataSource" rootEntity="false"
      processor="FileListEntityProcessor"
      baseDir="C:/docs" fileName="*.pdf"
      recursive="true">
      <field column="fileAbsolutePath" name="filePath" />
      <field column="fileSize" name="size" />
      <field column="fileLastModified" name="lastModified" />
      <entity name="tika-test" processor="TikaEntityProcessor"
        url="${pdfs.fileAbsolutePath}" format="text" dataSource="binfileDataSource">
        <field column="Author" name="author" meta="true"/>
        <field column="title" name="title" meta="true"/>
        <field column="text" name="text" />
      </entity>
    </entity>
  </document>
```

图 2-19 批量导入多个 PDF 配置示例

其他配置跟上一个示例保持不变。首先需要解释下为什么要定义两个 `DataSource` 以及这两个 `DataSource` 的区别。`FileDataSource` 用于将文件或文件夹转换成 Java 的 `Reader` 对象，`FileListEntityProcessor` 处理器需要接受此对象进行后续解析处理。而 `TikaEntityProcessor` 处理器需要接收 Java 里的 `InputStream` 输入流进行后续的 PDF 元数据和 PDF 文本内容提取工

作，所以你会看到第一个 entity 需要指定 `dataSource = "ileDataSource"`，而第二个 entity 需要指定 `dataSource = "binFileDataSource"`，理解 `fileDataSource` 和 `binFileDataSource` 的区别最直观的方式就是观摩两者的源码，如图 2-20 和图 2-21 所示。

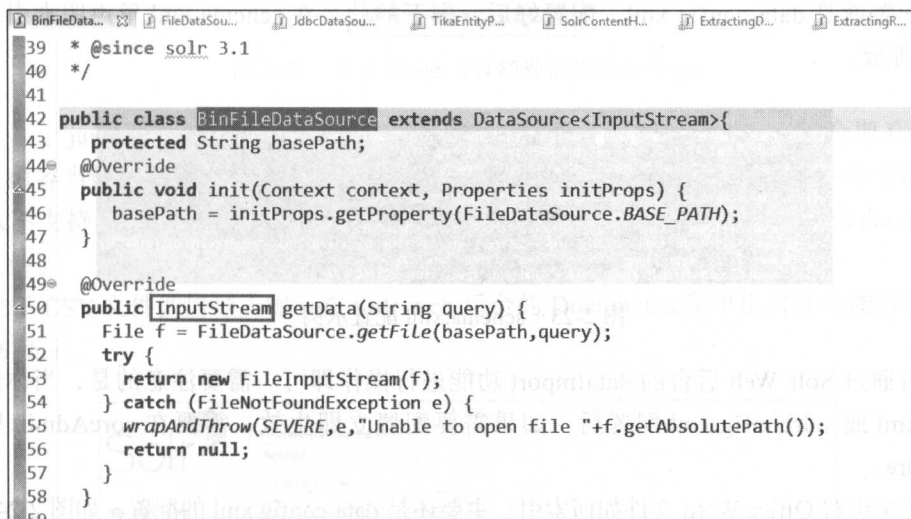


```

78  * <p>
79  * <b>It is the responsibility of the calling method to properly close the
80  * returned Reader</b>
81  * </p>
82  */
83  @Override
84  public Reader getData(String query) {
85      File f = getFile(basePath, query);
86      try {
87          return openStream(f);
88      } catch (Exception e) {
89          wrapAndThrow(SEVERE, e, "Unable to open File : "+f.getAbsolutePath());
90          return null;
91      }
92  }

```

图 2-20 FileDataSource 部分源码截图



```

39  * @since solr 3.1
40  */
41
42  public class BinFileDataSource extends DataSource<InputStream>{
43      protected String basePath;
44      @Override
45      public void init(Context context, Properties initProps) {
46          basePath = initProps.getProperty(FileDataSource.BASE_PATH);
47      }
48
49      @Override
50      public InputStream getData(String query) {
51          File f = FileDataSource.getFile(basePath, query);
52          try {
53              return new FileInputStream(f);
54          } catch (FileNotFoundException e) {
55              wrapAndThrow(SEVERE, e, "Unable to open file "+f.getAbsolutePath());
56              return null;
57          }
58      }
59  }

```

图 2-21 BinFileDataSource 类部分源码截图

为了更精细地控制 `FileListEntityProcessor` 处理器对文件的处理，比如你可能不需要处理文件夹下的所有文件，Solr 提供了 `fileName` 属性来限制文件名，它接收的是一个正则表达式。这里再次声明 `fileName` 是一个正则表达式，所以当你 `fileName = "*.pdf"` 这样配置时会报错，因为 “.” 点号在正则表达式里是一个特殊字符，你需要对 “.” 点号进行转义，况且 `*.pdf` 并不是一个合法的正则表达式，要表示所有文件名以 `.pdf` 结尾的文件的正则表达式正确写法应该是 `*\\.pdf`。`baseDir` 表示你需要对哪个文件夹下的文件进行批量索引，`processor` 是用来生成 Entity 的处理器，所谓的 Entity 其实就是 `entityProcessor` 生成的一个 Map 结构，



Map 的 key 就是你配置的 field 映射里的 column 属性值，不同的 entityProcessor 处理器会生成不同的 key，但每个 entityProcessor 处理器生成的 key 即 column 默认是内部事先定义好的。比如 FileListEntityProcessor 处理器会根据指定的文件夹生成多个 Entity，且生成的 Entity 会包含 fileAbsolutePath，fileSize，fileLastModified，fileName 这几个数据项，即 field 映射里的 column 值。而 TikaEntityProcessor 处理器生成的 Entity 会包含 Author、Title、Publisher、Language、Content 等几个数据项。Solr 内部其实实现了好几种 Entity-Processor 来满足常见的文件导入需求，具体如图 2-22 所示。

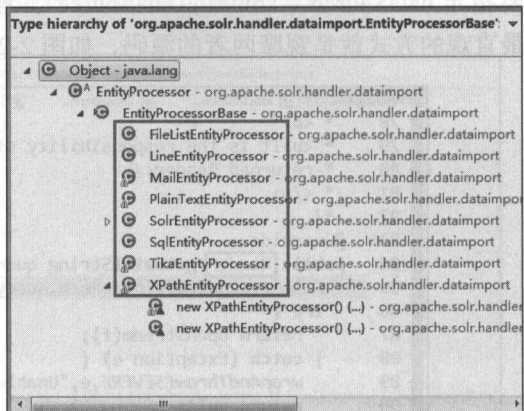


图 2-22 EntityProcessor 实现类

关于其他 EntityProcessor 处理器的使用会在后面的示例陆续介绍。pdf 文件批量导入的关键配置就是 data-config.xml，配置好后，剩下的就是在 schema.xml 里声明索引域，如图 2-23 所示。

```
<field name="text" type="text" indexed="true" stored="true" omitNorms="true" multiValued="false"/>
<field name="content" type="text" indexed="true" stored="true" omitNorms="true" multiValued="false"/>
<field name="author" type="string" indexed="true" stored="true" />
<field name="title" type="string" indexed="true" stored="true" />
<field name="dept" type="string" indexed="true" stored="true" />
<field name="fileName" type="string" indexed="true" stored="true" />
<field name="filePath" type="string" indexed="true" stored="true" required="false" multiValued="false" />
<field name="size" type="long" indexed="true" stored="true" />
<field name="lastModified" type="cdate" indexed="true" stored="true" />
```

图 2-23 schema.xml 配置示例

然后通过 Solr Web 后台的 dataImport 功能进行操作即可。需要注意的是，当你修改了 schema.xml 或 solrconfig.xml 配置后，如果需要配置立即生效，需要在 coreAdmin 里重新 reload core。

接下来讲解 Office Word 文件如何索引？主要还是 data-config.xml 的配置，如图 2-24 所示。

```
<dataConfig>
  <dataSource name="fileDataSource" type="FileDataSource" />
  <dataSource name="binFileDataSource" type="BinFileDataSource" />
  <document>
    <entity name="pdfs" dataSource="fileDataSource" rootEntity="false"
      processor="FileListEntityProcessor"
      baseDir="C:/docs" fileName=".*\\.(doc)|(docx)|(pdf)|(xls)|(xlsx)|(ppt)|(pptx)"
      recursive="true">
      a
    <entity name="tika-test" processor="TikaEntityProcessor"
      url="{pdfs,fileAbsolutePath}" format="text" dataSource="binFileDataSource" onError="skip">
      <field column="Author" name="author" meta="true"/>
      <field column="Title" name="title" meta="true"/>
      <field column="Text" name="text" />
    </entity>
  </entity>
</document>
```

图 2-24 导入 office-word 时 data-config.xml 配置示例



先用 `FileListEntityProcessor` 处理器遍历文件夹，将符合 `fileName` 文件名规则的文件转成一个 `Entity` 对象，然后使用 `TikaEntityProcessor` 处理器对提取到的文件进行解析。相同的参数上面已经解释过了，这里不再赘述了。`rootEntity` 参数在默认情况下为 `false`。`Document` 元素下就是根实体了，如果没有根实体，直接在实体下面的实体将会被看作根实体。对于根实体对应的数据库中返回的数据的每一行，Solr 都将生成一个 `Document`。`onError` 参数表示当对 `entity` 进行数据解析发生异常时怎么处理，默认是 `"abort"`，即中断 `entity` 解析工作，`"skip"` 表示跳过当前文档不处理了，`"continue"` 表示对错误视而不见。至此 Word 文件导入 Solr 也完成了。需要注意的是，通过我亲测发现，Solr 5.3.1 版本只支持 doc 文件导入，不支持 docx 文件导入。至于如何索引 Excel 文件同理，但还是提醒一句：对于 Excel 文件，Tika 不提取 `title` 域，所以 `field` 映射里可以不需要 `title` 域。同 Word 文件一样，Solr 5.3.1 版本不支持 `xlsx` 文件导入，请知晓！另外导入 Excel 文件时，需要在你的 `core\lib` 目录下额外添加 3 个 jar 包，如图 2-25 所示。

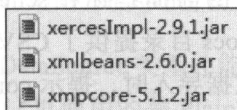


图 2-25 导入 Word 文件额外依赖的 3 个 jar

这 3 个 jar 包在 `solr-5.3.1\contrib\extraction\lib` 目录下可以找到。Tika 只会把 Excel 文件的所有内容当作一个文本存入 `text` 域，如果你想实现 Excel 里每一行都生成一个 `Document`，则默认不支持。这时候建议在 `solrJ` 客户端使用 POI 或者直接模仿 Solr 使用 Tika 去解析富文本文件。

对于 CSV 文件如何导入呢？在 Solr web 后台的 `Documents` 菜单里对此功能有提供，如图 2-26 所示。

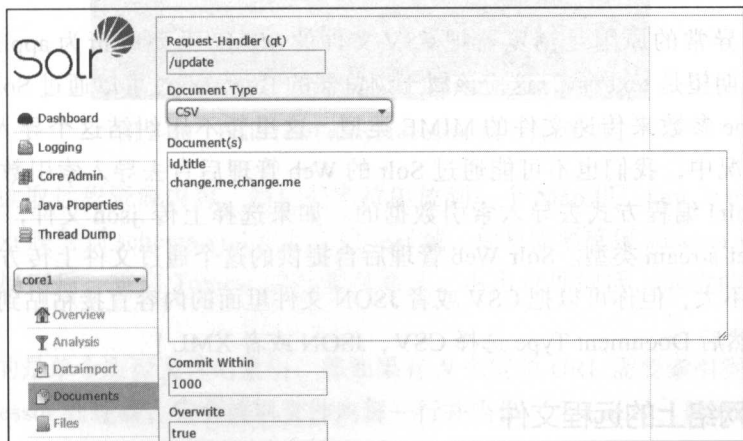


图 2-26 CSV 格式数据导入

Documents 文本框里填写 CSV 格式的数据，第一行是定义列，列名之间使用英文逗号分隔，从第二行开始是每列的数据，根据第一列的列号对号入座，值之间也是使用英文逗号分隔，你提交后，Solr 会以你第一行定义的列名去添加域，所以在提交之前你必须确保已经在 schema.xml 中提前定义了该域。如图 2-26 所示，你必须提前定义 id 和 title 域。CommitWithin 参数表示当前提交必须在指定的毫秒数内执行完成，若未完成，则中止本次提交，从等待队列里弹出另一个新的提交继续处理。若当前提交正在处理中，这时你又发起了一个新的提交，若当前提交的处理耗时仍在 CommitWithin 限定时间范围内，则后续发起的其他新的提交全部阻塞进入等待队列，要么等待当前提交处理完成，要么等待当前提交处理耗时超出 CommitWithin 参数限定的毫秒数（即处理超时）。CommitWithin 参数若不设置，默认值为 -1，-1 表示不做超时限制。具体请参阅 CommitTracker 类源码。

overwrite 参数表示是否根据 Document 的主键去覆盖之前添加的 Document，如果两者 Document 主键相同且 overwrite 参数设置为 true，则新添加的 Document 会覆盖旧的 Document，默认值为 true。Overwrite=true 生效的前提是你在 schema.xml 里指定了 <uniqueKey>。在 Solr zip 压缩包 example\exampledocs 目录提供了 CSV 测试文件，不过可惜的是，Solr 的 Web 后台直接上传 CSV 文件进行数据导入时，提示 content-type 不在 Solr 支持范围内，如图 2-27 所示。

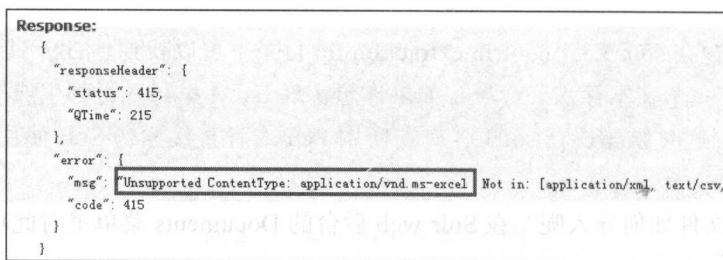


图 2-27 CSV 文件导入提示异常

出现这种异常的原因是浏览器把 CSV 文件的 MIME 类型解析为 application/vnd.ms-excel，而 Solr 期望是 text/csv，这应该属于浏览器的 Bug，不过可以通过 SolrJ 编程的方式指定 stream.type 参数来传递文件的 MIME 类型。这里暂不用纠结这个导入功能的使用，毕竟在实际情况下，我们也不可能通过 Solr 的 Web 管理后台去导入索引数据，大部分情况还是通过 SolrJ 编程方式去导入索引数据的。如果选择上传 .json 文件，会提示不支持 application/octet-stream 类型。Solr Web 管理后台提供的这个通过文件上传方式导入索引的功能虽然用处不大，但你可以把 CSV 或者 JSON 文件里面的内容直接粘贴到 Documents 多行文本框里，然后 Document Type 选择 CSV、JSON 或者 XML！

## 2.2.4 索引网络上的远程文件

有时候你需要索引的文件可能不在本地，而是存在于互联网上。当然，你可以写个爬

虫将其抓取到本地硬盘然后借助上一章节介绍的方式去索引。其实 Solr 已经内置了 URLDataSource 支持直接获取远程资源进行索引。

URLDataSource 的核心代码如图 2-28 所示。

URLConnection 类我想大家都不陌生了吧，直接通过 URLConnection 类建立与远程资源的连接并将其转换为 java.io.Reader 对象。然后在 data-config.xml 中配置 URLDataSource 数据源，如图 2-29 所示。

```
public Reader getData(String query) {
    URL url = null;
    try {
        if (URIMethod.matcher(query).find()) url = new URL(query);
        else url = new URL(baseUrl + query);

        LOG.debug("Accessing URL: " + url.toString());

        URLConnection conn = url.openConnection();
        conn.setConnectTimeout(connectionTimeout);
        conn.setReadTimeout(readTimeout);
```

图 2-28 URLDataSource 类部分源码

```
<document>
  <dataSource name="urlDataSource" type="URLDataSource" />
  <entity processor="PlainTextEntityProcessor" name="onlineTxtFile" url="http://tech.163.
    com/15/0527/16/AQKTMJB0800915BF.html" dataSource="urlDataSource">
    <field column="plaintext" name="text"/>
  </entity>
</document>
```

图 2-29 URLDataSource 配置示例

关键点是 dataSource 必须是 urlDataSource 类型才能加载远程资源文件，url 表示一个远程资源文件的访问 URL。processor 这里使用的是 PlainTextEntityProcessor 处理器，核心代码如图 2-30 所示。

```
59 StringWriter sw = new StringWriter();
60 char[] buf = new char[1024];
61 while (true) {
62     int len = 0;
63     try {
64         len = r.read(buf);
65     } catch (IOException e) {
66         IOUtils.closeQuietly(r);
67         wrapAndThrow(SEVERE, e, "Exception reading url : " + url);
68     }
69     if (len <= 0) break;
70     sw.append(new String(buf, 0, len));
71 }
72 Map<String, Object> row = new HashMap<>();
73 row.put(PLAIN_TEXT, sw.toString());
74 ended = true;
75 IOUtils.closeQuietly(r);
76 return row;
77 }
78
79 public static final String PLAIN_TEXT = "plaintext";
```

图 2-30 PlainTextEntityProcessor 核心代码截图

它会逐行读取远程资源内容，最后把字符串放到一个 Map 里，key 为 plaintext。然后我们把 plaintext 映射到 schema.xml 里定义的 text 域，你只需要确保 schema.xml 中存在 text 域即可，配置搞定后，重启 Tomcat 进行索引测试，不出意外的话，远程的资源文件会被 Solr 索引成功。

上面介绍的是单个资源文件的索引，那如果有  $N$  个资源 URL 需要索引呢？Solr 提供了 LineEntityProcessor 处理器，它会读取文件内每一行并当作一个 Entity，部分源码如下所示：

```
public Map<String, Object> nextRow() {
```

```

if (reader == null) {
    reader = new BufferedReader((Reader) context.getDataSource().getData(url));
}
String line;
while ( true ) {
    try {
        line = reader.readLine();
    }
    catch (IOException exp) {
        throw new DataImportHandlerException(DataImportHandlerException.SEVERE,
            "Problem reading from input", exp);
    }
    if (line == null) {
        closeResources();
        return null;
    }
    if (acceptLineRegex != null && ! acceptLineRegex.matcher(line).find()) continue;
    if (skipLineRegex != null && skipLineRegex.matcher(line).find()) continue;
    Map<String, Object> row = new HashMap<>();
    row.put("rawLine", line);
    return row;
}
}

```

首先它需要一个 reader 对象，所以很明显 LineEntityProcessor 处理器需要一个 FileDataSource 数据源，URLDataSource 数据源虽然也可以生成 java.io.Reader 对象，但它主要用于加载远程资源，这里为了批量加载远程的网络资源，我们需要把资源 URL 写入一个文本文件里，一个 URL 独占一行。LineEntityProcessor 会读取每一行的内容，并将每一行的内容以 urawLine 为 Key 存入 Map 结构。为了过滤一些干扰数据，这里还提供了两个额外的参数：acceptLineRegex 和 skipLineRegex。如果读取到的当前行内容为 Null，则表明已经读取到文件末尾，直接跳出循环。所以由阅读源码我们可以得知，最后一行不能为空行，否则会直接返回 null，导致前面的所有数据都读取失败。最后，我们的 data-config.xml 可以这样配置，如图 2-31 所示。

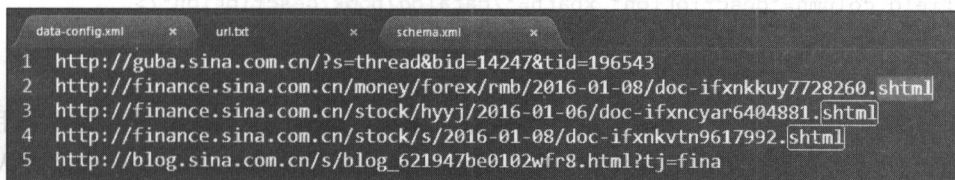
```

<dataSource name="fileDataSource" type="FileDataSource" />
<dataSource name="urlDataSource" type="URLDataSource" />
<document>
  <entity name="line"
    processor="LineEntityProcessor"
    acceptLineRegex="^.*\\.shtml$"
    url="C:/docs/url.txt"
    rootEntity="false"
    dataSource="fileDataSource"
  >
    <entity processor="PlainTextEntityProcessor" name="onlineTxtFile"
      url="{line.rawLine}" dataSource="urlDataSource">
      <field column="plainText" name="text"/>
    </entity>
  </entity>
</document>

```

图 2-31 批量导入远程资源文件配置示例

`acceptLineRegex`：表示当前行内容是否接受处理的正则表达式，只有符合此正则表达式的行才会被处理，同理还有 `skipLineRegex`，表示不符合此正则表达式的行将会被跳过不予处理。`url` 参数表示文件的路径。然后需要在 `C:/docs` 目录下新建 `url.txt` 文件，往里面放置一个 `url`，这里我从 `sina` 找了几个新闻详情页的 `url`，如图 2-32 所示。



```

1 http://guba.sina.com.cn/?s=thread&bid=14247&tid=196543
2 http://finance.sina.com.cn/money/forex/rmb/2016-01-08/doc-ifyxkky7728260.shtml
3 http://finance.sina.com.cn/stock/hyyj/2016-01-06/doc-ifyxncyar6404881.shtml
4 http://finance.sina.com.cn/stock/s/2016-01-08/doc-ifyxkvtn9617992.shtml
5 http://blog.sina.com.cn/s/blog_621947be0102wfr8.html?tj=fina
  
```

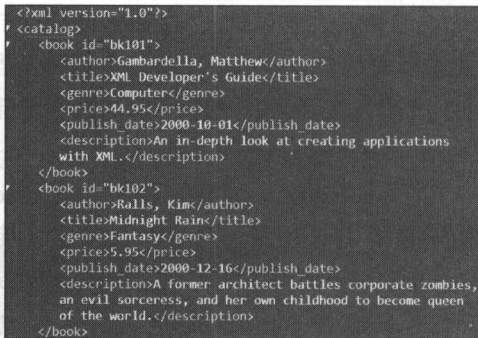
图 2-32 测试 URL

然后你需要在 `schema.xml` 里定义 `text` 域，通过 Solr Web 后台的 `dataImport` 进行导入，不出意外的话，会提示成功导入了 3 条索引数据，因为上面我们的 `acceptLineRegex` 正则表达式限定了只处理以 `.shtml` 结尾的 URL 链接。

## 2.2.5 索引 XML 文件

假定你有类似这样一个 XML 文件，如图 2-33 所示。

通常你肯定是希望把每个 `book` 元素当作一个 Document 进行导入，而不是把 XML 文件的全部内容导入一个域，Solr 提供了 `XPathEntityProcessor` 使用 `XPath` 表达式来提取 XML 内容，使用 `XPathEntityProcessor` 的前提是你要熟悉 `XPath` 表达式，如果对 `XPath` 不熟悉，请先去 `W3School` 学习了解。上面的 `book.xml` 是我通过 Google 随便搜索到的，大家自己找个方便测试的 `xml` 文件即可，如果实在找不到合适的测试 XML，请从这个地址下载：



```

<?xml version="1.0"?>
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price>44.95</price>
    <publish date>2000-10-01</publish date>
    <description>An in-depth look at creating applications with XML.</description>
  </book>
  <book id="bk102">
    <author>Ralls, Kim</author>
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish date>2000-12-16</publish date>
    <description>A former architect battles corporate zombies, an evil sorceress, and her own childhood to become queen of the world.</description>
  </book>
</catalog>
  
```

图 2-33 XML 测试文件

<https://dumps.wikimedia.org/enwiki/>

具体配置如下所示：

```

<dataSource name="fileDataSource" type="FileDataSource" encoding="UTF-8" />
<document>
  <entity name="books" processor="XPathEntityProcessor"
    dataSource="fileDataSource" stream="true"
    forEach="/catalog/book"
    url="C:/docs/books.xml"
    transformer="DateFormatTransformer">
    <field column="id" xpath="/catalog/book/@id" multiValued="false" flatten="true"/>
  
```



```

<field column="title" xpath="/catalog/book/title"/>
<field column="genre" xpath="/catalog/book/genre"/>
<field column="author" xpath="/catalog/book/author"/>
<field column="price" xpath="/catalog/book/price"/>
<field column="publishDate" xpath="/catalog/book/publish_date"
      dateTimeFormat="yyyy-MM-dd"/>
<field column="description" xpath="/catalog/book/description"/>
</entity>
</document>

```

稍微对上面的配置几个比较重要的参数做下解释，`stream` 表示是否另起线程采用队列分批处理，当单个 xml 文件里包含几万行，比如 xml 中有 10000 个 `<book>` 元素要导入，那建议设置 `stream=true` 以提高导入性能。观摩下 `XPathEntityProcessor` 类的源码你就明白了。`Transformer` 表示数据格式转换器，这里使用了 `DateFormatTransformer`，很明显，它是用来进行日期格式转换的，这里我们用它将日期字符串转换成 Java 里的 `util.Date`，因为在 `schema.xml` 中 `publishDate` 域的域类型设置为 `tdate`，而 Solr 里的 `cndate` 域类型对应 Java 里的 `java.util.Date`，如果日期域类型定义为 `string`，那就不需要指定 `transformer` 转换器了！下面的 `field` 就是要提取的数据项定义，`column` 表示提取出来的数据项的 `key`，Solr 默认也是取 `column` 属性值作为域名称的，这点隐含规则请知晓。后面的 `xpath` 是你提取当前数据项需要使用到的 XPath 表达式。至于 XPath 表达式怎么写这里就不做介绍了。为了演示，我在第一个 `field` 元素里故意添加了 `multiValued` 和 `flatten` 两个参数，默认值均为 `false`。`multiValued` 表示这个域是个多值域，`flatten` 表示如果当前 `xpath` 表达式定位到的元素下如果还有其他子元素，是否需要下面所有子元素的文本全部拼接在一起，然后追加到当前元素文本的后面，拼接后的整体作为当前域的域值。

在 `publishDate` 域我配置了一个 `dateTimeFormat` 参数，用于指定日期格式，对这个参数务必要引起注意，错误的配置会引起整个 Document 导入失败。`<field>` 元素还有一个 `commonField` 参数，它接收布尔值 (`true/false`)，如果设置为 `true`，即表示这是一个通用域，即如果其他 Document 没有包含此域，则会自动添加上此域并自动复制域值，不设置默认值为 `false`。



**注意** 是否导入成功务必以提示的 `Processed` 数量为准。

在本章前面的章节中，我提到过 Solr 规定的一种特定的 XML 格式，如图 2-34 所示。

图 2-34 中的 xml 在 `solr-5.3.1/example/exampledocs` 目录下可以找到，对于这种格式的 xml，Solr 也是支持导入的。在 `data-config.xml` 配置上稍微简洁点，因为是 Solr 规范的格式，所以不需要提供 `xpath` 表达式告诉 Solr

```

<add>
<doc>
  <field name="id">USD</field>
  <field name="name">One Dollar</field>
  <field name="manu">Bank of America</field>
  <field name="manu id s">boa</field>
  <field name="cat">currency</field>
  <field name="features">Coins and notes</field>
  <field name="price_c">1,USD</field>
  <field name="inStock">true</field>
</doc>

```

图 2-34 Solr 支持的规范 XML 数据



如何去提取数据，具体配置如下所示：

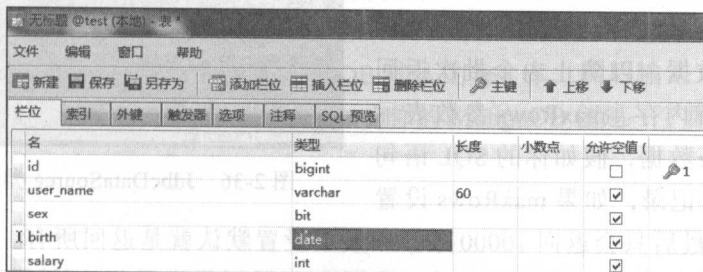
```
<dataSource name="fileDataSource" type="FileDataSource" encoding="UTF-8" />
<document>
<entity name="money"
  processor="XPathEntityProcessor"
  dataSource="fileDataSource"
  stream="true"
  useSolrAddSchema="true"
  forEach="/catalog/book"
  url="C:/docs/money.xml"
  transformer="DateFormatTransformer">
</entity>
</document>
```

重点就是 useSolrAddSchema 参数，这时候就不再需要配置 field 映射了，记得在 schema.xml 里提前定义好域，这里我就不贴 shcema.xml 的配置了，自己参照 xml 里 field 的 name 属性值和文本类型斟酌下就知道了。

**提示** 修改了 schema.xml 或 solrconfig.xml 或 data-config.xml 等几个配置文件记得重新加载一下你的 Core。

## 2.2.6 从数据库中导入数据至 Solr

可能你的 Solr 索引数据来自于关系型数据库，所以你可能会有这样的需求：如何把 MySQL 数据库表里的数据导入 Solr。Solr 为此提供了 JdbcDataSource。假定你有类似这样的一张表，如图 2-35 所示。



名	类型	长度	小数点	允许空值 (
id	bigint			<input type="checkbox"/> 1
user_name	varchar	60		<input checked="" type="checkbox"/>
sex	bit			<input checked="" type="checkbox"/>
birth	date			<input checked="" type="checkbox"/>
salary	int			<input checked="" type="checkbox"/>

图 2-35 自定义的 MySQL 测试表

要读取数据库显然你需要导入数据库驱动 jar 包，请下载 MySQL 的驱动 jar 包并复制到当前 Core 的 lib 目录下。如果使用的是 Oracle 数据库，那显然你需要导入 Oracle 的驱动 jar 包。重点还是配置我们的 data-config.xml：

```
<dataSource name="jdbcDataSource" type="JdbcDataSource" driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=u
```

```
tf-8" user="root" password="123"/>
<document>
<entity dataSource="jdbcDataSource" name="user" query="select * from user">
<field column="id" name="id"/>
<field column="user_name" name="userName"/>
<field column="sex" name="sex"/>
<field column="birth" name="birth"/>
<field column="salary" name="salary"/>
</entity>
</document>
```

name 表示给你的数据源起个别名，便于 entity 元素引用，type 就是 Solr 内置的数据源的类名，driver 就是驱动类的完整包路径，url 就是你的 jdbc 连接 url，这里需要注意的是，url 不要包含 html 中的特殊字符，以下 5 个字符请不要出现在 url 中，具体如表 2-1 所示。

表 2-1 HTML 中的特殊字符

Entity	字符	描述
&lt;	<	小于号
&gt;	>	大于号
&amp;	&	& 符号
&apos;	'	单引号
&quot;	"	双引号

如果这些字符已经存在，请使用它的实体引用替换，虽然这个问题已经被大家问过好多次了，但我还是要提醒下。dataSource 元素中还有两个比较重要的参数 batchSize 和 maxRows，如图 2-36 所示。

batchSize 表示 JDBC 从数据库一个批次提取多少条数据，以防止一个批次返回数据量过大撑爆内存。maxRows 参数表示最多返回多少条数据，假如你的 SQL 语句返回了 100 万条记录，如果 maxRows 设置

```
public class JdbcDataSource extends
    DataSource<Iterator<Map<String, Object>>> {
    private static final Logger LOG = LoggerFactory.getLogger(JdbcDataSource.class);
    protected Callable<Connection> factory;
    private long connLastUsed = 0;
    private Connection conn;
    private Map<String, Integer> fieldNameVsType = new HashMap<>();
    private boolean convertType = false;
    private int batchSize = FETCH_SIZE;
    private int maxRows = 0;
```

图 2-36 JdbcDataSource 类部分源码截图

为 10000，那么最后只会返回 10000 条，一般不设置默认就是返回所有。此外 dataSource 还有个 convertType 参数，接收布尔值，表示是否需要指定数据库表字段的类型，如果 convertType 设置为 type，那么下面的 <field> 映射需要添加 type 参数，例如：

```
<field column="id" name="id" type="long"/>
```

之所以配置 type 参数是为了便于 resultSet.getLong() 直接获取字段值，了解 JDBC 的应该都可以很快了解。type 参数的可选值范围其实源码里已经写得很清楚，如图 2-37 所示。

看了上面 JdbcDataSource 类的部分代码截图，已经不言自明了吧。其实 JdbcDataSource

还支持 JNDI 数据源，你可以在 `<dataSource>` 元素里指定 `jndiName` 配置参数，参数值即 JNDI 数据源的别名。JNDI 数据源如何配置请读者自行查阅。

`entity` 的 `name` 表示给你的 `entity` 起个别名，`query` 即需要执行的 SQL 语句，该 SQL 语句执行后返回的结果集都将会被导入 Solr。下面的 `<field>` 部分就是数据库表字段和 Solr 的域名称之间的映射，有点类似 Hibernate 里数据库表字段名与类属性名称之间的映射。

然后照常你需要在 `schema.xml` 中定义域，如图 2-38 所示。

```
for (Map<String, String> map : context.getAllEntityFields()) {
    String n = map.get(DataImporter.COLUMN);
    String t = map.get(DataImporter.TYPE);
    if ("int".equals(t) || "integer".equals(t))
        fieldNameVsType.put(n, Types.INTEGER);
    else if ("slong".equals(t) || "long".equals(t))
        fieldNameVsType.put(n, Types.BIGINT);
    else if ("float".equals(t) || "sfloat".equals(t))
        fieldNameVsType.put(n, Types.FLOAT);
    else if ("double".equals(t) || "sdouble".equals(t))
        fieldNameVsType.put(n, Types.DOUBLE);
    else if ("date".equals(t))
        fieldNameVsType.put(n, Types.DATE);
    else if ("boolean".equals(t))
        fieldNameVsType.put(n, Types.BOOLEAN);
    else if ("binary".equals(t))
        fieldNameVsType.put(n, Types.BLOB);
    else
        fieldNameVsType.put(n, Types.VARCHAR);
}
```

图 2-37 type 参数可选值范围

```
<field name="userName" type="string" indexed="true" stored="true" omitNorms="true"/>
<field name="sex" type="boolean" indexed="true" stored="true" omitNorms="true"/>
<field name="birth" type="cndate" indexed="true" stored="true" omitNorms="true"/>
<field name="salary" type="int" indexed="true" stored="true" omitNorms="true"/>
```

图 2-38 Schema.xml 配置 field

然后重新加载你的 Core，最后在 `dataImport` 中执行索引导入操作即可。有关关系型数据库数据导入 Solr 就介绍这么多，这里是以 MySQL 为例，其他关系型数据库的导入方式与此类似，只是导入的驱动 jar 包、JDBC 连接 URL、驱动类、执行的查询 SQL 语句语法可能会有点不一样。

与关系型数据库对应的还有 NoSQL 数据库，我们常用的应该是 MongoDB，这里我就以 MongoDB 为例，讲解如何将 MongoDB 里的数据导入 Solr。

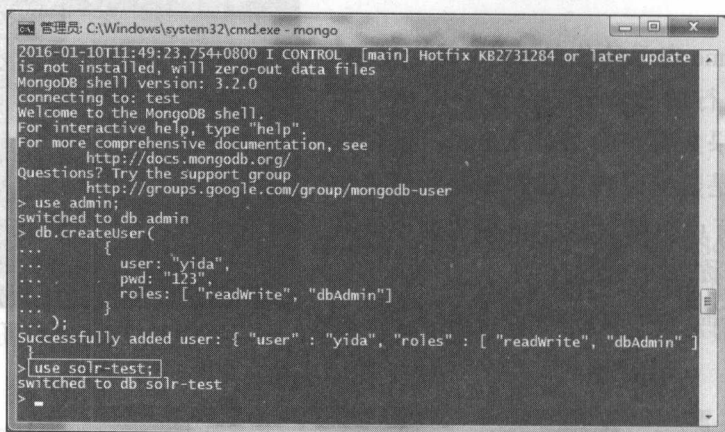
首先你需要安装 MongoDB 数据库环境，这里就不对 MongoDB 如何安装做详细介绍了。如果你的 MongoDB 数据库环境已经准备就绪，那么就随我一起开始 MongoDB 数据导入 Solr 之旅吧。

首先请输入 `mongo` 命令连接到 MongoDB 默认的 `test` 库，然后输入 `use admin` 切换到 `admin` 管理员数据库，然后通过 `db.createUser` 为你的 MongoDB 创建一个管理员账号：

```
db.createUser(
{
    user: "yida",
    pwd: "123",
    roles: [ "readWrite", "dbAdmin" ]
});
```

**注意** MongoDB 不同版本的操作命令会有一些区别，这里我是以 MongoDB-3.2.0 版本为例进行演示，至于其他版本是否会完全兼容本小节涉及到的所有 MongoDB 操作命令，请自行查阅官方文档进行确认。

上面我们创建一个账号为 yida 密码为 123 的管理员账号，然后通过 use 命令创建一个测试数据库，如图 2-39 所示。

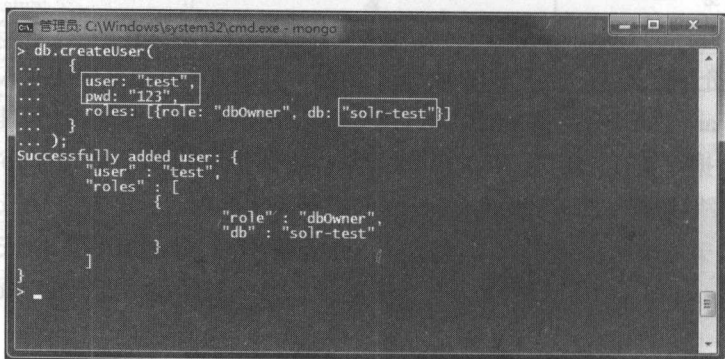


```

C:\Windows\system32\cmd.exe - mongo
2016-01-10T11:49:23.754+0800 I CONTROL [main] Hotfix KB2731284 or later update
is not installed, will zero-out data files
MongoDB shell version: 3.2.0
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
> use admin;
switched to db admin
> db.createUser(
...
  {
    user: "yida",
    pwd: "123",
    roles: [ "readwrite", "dbAdmin" ]
  }
);
Successfully added user: { "user" : "yida", "roles" : [ "readwrite", "dbAdmin" ] }
> use solr-test;
switched to db solr-test
>
  
```

图 2-39 创建 MongoDB 用户和数据库

为 solr-test 数据库创建一个测试账号，并为其赋予权限，如图 2-40 所示。



```

C:\Windows\system32\cmd.exe - mongo
> db.createUser(
...
  {
    user: "test",
    pwd: "123",
    roles: [ {role: "dbOwner", db: "solr-test"} ]
  }
);
Successfully added user: {
  "user" : "test",
  "roles" : [
    {
      "role" : "dbOwner",
      "db" : "solr-test"
    }
  ]
}
>
  
```

图 2-40 创建 MongoDB 测试用户

创建一个 name 为 book 的 Collection，如图 2-41 所示。

通过 `db.{collectionName}.insert` 命令往刚刚创建的 collection 里插入几条测试数据，{collectionName} 泛指你的 collection 名称，这里指代 book，如图 2-42 所示。

到这里 MongoDB 测试数据准备完毕，Java 想要连接 MongoDB 数据库并读取其中数据，需要导入 MongoDB 的驱动包，在网络中搜索“MongoDB Java Driver Maven”可以获取该驱动包。为了避免出现意外，请尽量使你的 MongoDB 驱动包和 MongoDB Server 版本保持一致。因为我安装的 MongoDB Server 版本是 3.2.0，所以这里下载的是 mongo-java-driver-3.2.0.jar，然后将其复制到你导入 Core 的 lib 目录下。Solr 官方并没有提供 NOSqlDataSource 数据源类来读取 MongoDB，你可以在网络中搜索“solr mongo dataimport github”获取相关扩展源码以及 jar 包。

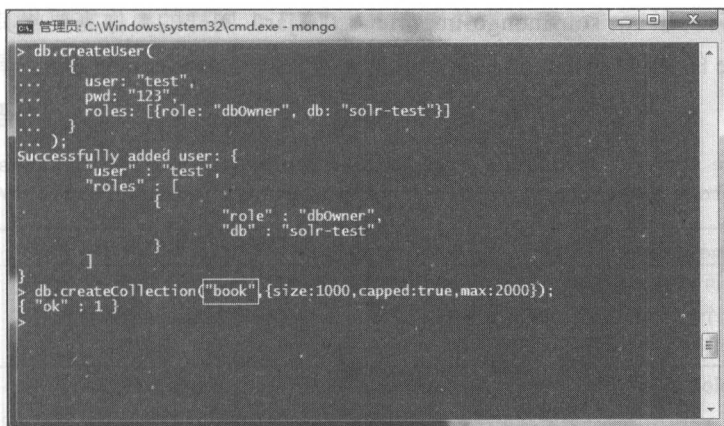


图 2-41 创建一个 MongoDB 的测试 Collection

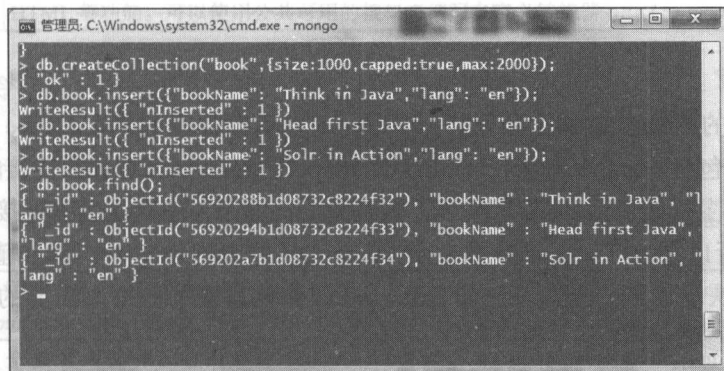


图 2-42 往 MongoDB 的 Collection 中 insert 数据

solr-mongo-import-version.jar github 下载地址如下:

<https://github.com/james75/SolrMongoImporter>

← solr-mongo-import 的 jar 包下载如图 2-43 所示。

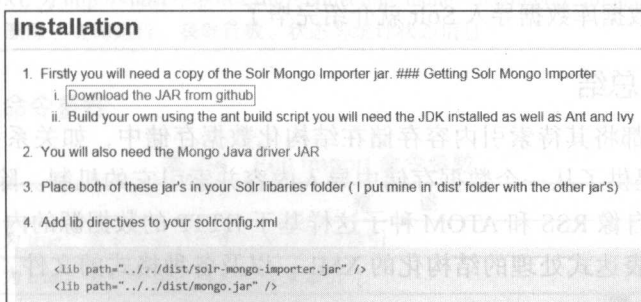


图 2-43 Solr-mongo-importer.jar 下载示意图



下载后你将得到一个 solr-mongo-importer-1.0.0.jar, 同样的, 你需要将其复制到你需要导入的目标 Core 的 lib 目录下。然后重点就是配置我们的 data-config.xml, 具体配置如下代码所示:

```
<dataSource name="mongoDataSource" type="MongoDataSource" database="solr-test"
  host="localhost" port="27017" username="test" password="123"/>
<document>
  <entity name="book"
    processor="MongoEntityProcessor"
    query="{ 'lang': 'en' }"
    collection="book"
    dataSource="mongoDataSource"
    rootEntity="true"
    onError="skip"
    transformer="MongoMapperTransformer">
    <field column="bookName" name="bookName" mongoField="bookName"/>
    <field column="lang" name="lang" />
  </entity>
</document>
```

dataSource 的配置参数如上所示, 其中 name 表示数据源的别名; type 即数据源的类名, 省去了默认包名 org.apache.solr.handler.dataimport; database 即你要连接的数据库名称; host 即你要连接主机的 IP 地址或者域名, 默认是 localhost, 即默认会去连接本机, 这里是为了演示目的所以加上了。port 参数表示 MongoDB 监听的端口号, 默认值为 27017, 即 MongoDB 的默认端口号。username 和 password 即你要连接的数据库需要的账号密码, 这两个参数是必需参数。

entity 里 processor 参数指定使用 MongoEntityProcessor 这没什么好说的, query 即 MongoDB 里的查询语法, 懂 MongoDB 的, 我想也无须多言。collection 参数表示你要查询的 Collection 的名称, dataSource 参数引用上面 <dataSource> 元素配置的数据源的别名。transformer 参数用于解决在 MongoDB 返回的结果集里的 field 名称跟原始 collection 里的 field 不一致的情况下, 需要添加 mongoField 再对 column 与返回结果集里的 field 做个映射。

同理你需要在 schema.xml 定义好相关域, 这里即 bookName 和 lang 两个域, 然后执行导入。到此有关数据库数据导入 Solr 就介绍完毕了。

## 2.2.7 Solr DIH 总结

很多搜索应用都将其待索引内容存储在结构化数据存储中, 如关系型数据库。数据导入处理器 (DIH) 提供了从一个数据存储中导入内容并索引它的机制。除了关系型数据库, DIH 还可以索引来自像 RSS 和 ATOM 种子这样基于 HTTP 的数据源的内容、E-mail 仓库以及能够使用 XPath 表达式处理的结构化的 XML, 以及各种格式的文件, 比如 TXT、CSV、JSON、Word、PDF 等。

在使用 Solr DIH 的过程中, 主要涉及如下几种 DIH 术语, 为了更好地正确使用



Solr DIH 功能，我们有必要了解 Solr DIH 涉及的相关术语表示的详细含义，具体请查阅表 2-2。

## 1. DIH 术语

表 2-2 DIH 术语解释

术语	描 述
DataSource	定义 Solr 索引数据的来源，可以是数据库、HTTP 资源、文件
Entity	概念上，实体会被处理以生成包含多个字段的 Document 集合，它们（可以选择以多种不同方式进行转换）会被传递给 Solr 进行索引。对一个 RDBMS 数据源，一个实体是一个视图或者表，它可能会被一或多个 SQL 语句处理生成一系列具有一或多个列（字段）的行（文档）
EntityProcessor	一个实体处理器会完成整个从数据源抽取内容、转换并将其添加到 Solr 创建索引的工作。可以用自定义实体处理器扩展或替换 Solr 提供的默认处理器
Transformer	从实体中取得的每个字段的集都可以选择进行转换。这个过程可以修改字段、创建新字段或者从单个行生成多个行 / 文档。DIH 中内置了多个转换器，它们可以执行像修改日期、剥离 HTML 等功能。可以使用公共可用的接口来编写自定义转换器

## 2. DIH 命令

表 2-3 DIH 命令解释

命令	描 述
abort	取消一个正在进行的 dataimport 操作。其 URL 为 <code>http://&lt;host&gt;:&lt;port&gt;/solr/&lt;collection_name&gt;/dataimport?command=abort</code>
delta-import	用于增量导入和变更侦测。其命令 URL 为 <code>http://&lt;host&gt;:&lt;port&gt;/solr/&lt;collection_name&gt;/dataimport?command=delta-import</code> 。它支持和全量导入一样的，诸如 <code>clean/commit/optimize/debug</code> 之类的参数
full-import	全量导入操作，它的命令 URL 为 <code>http://&lt;host&gt;:&lt;port&gt;/solr/&lt;collection_name&gt;/dataimport?command=full-import</code> 。该命令会立即返回，其操作会启动一个新线程且其响应中的 <code>status</code> 属性应该显示为 <code>busy</code> 。该操作可能会比较耗时。全量导入时对 Solr 的查询操作不会进行阻塞。当一个全量导入命令被执行时，它会在位于 <code>conf/dataimport.properties</code> 的配置文件中存储操作的起始时间。这个存储的时间戳在 <code>delta-import</code> 操作被执行时使用。该命令可接收的参数列表，请参见《full-import 命令的参数》
reload-config	若配置有变更且你希望不用重启 Solr 就重新加载配置文件，可以运行此命令 <code>http://&lt;host&gt;:&lt;port&gt;/solr/&lt;collection_name&gt;/dataimport?command=reload-config</code>
status	其 URL 为 <code>http://&lt;host&gt;:&lt;port&gt;/solr/&lt;collection_name&gt;/dataimport?command=status</code> 。它会返回创建、删除、查询运行、获取行数、状态等统计状态信息

## 3. Full Import 命令参数

表 2-4 Full Import 命令参数

命令	描 述
clean	默认为 <code>true</code> 。表示在索引开始时是否清空索引
commit	默认为 <code>true</code> 。表示在操作后是否提交索引
debug	默认为 <code>false</code> 。以 <code>debug</code> 模式运行命令，用于交互式开发模式。注意在 <code>debug</code> 模式中，文档不会自动提交。若你希望运行 <code>debug</code> 模式，你也需要通过在请求中添加 <code>commit=true</code> 参数来提交索引

(续)

命令	描 述
entity	直接在配置文件的 <document> 标签的实体名称。使用它可以选择性地执行一或多个实体。可以一次性传入多个 entity 参数来执行多个实体。若没有该参数, 所有的实体都会被执行
optimize	默认为 true。表示在操作后是否优化索引

#### 4. propertyWriter 属性写入器

propertyWriter 元素定义了 delta 查询时使用哪种日期格式和地区。这是一个可选配置。直接在 DIH 配置文件的数据Config 元素下添加该元素, 配置示例如下, 可用参数如表 2-5 所示。

```
<propertyWriterdateFormat="yyyy-MM-dd HH:mm:ss" type="SimplePropertiesWriter"
directory="data" filename="my_dih.properties" locale="en_US" />
```

表 2-5 可用参数

参数	描 述
dateFormat	用于将日期转换为文本的 java.text.SimpleDateFormat, 默认为 "yyyy-MM-dd HH:mm:ss"
type	用于指定 propertyWriter 的实现类。在非 SolrCloud 环境使用 SimplePropertiesWriter, 在 SolrCloud 环境使用 ZKPropertiesWriter。若没有指定, 它会根据 SolrCloud 环境是否启用自动适配合适的实现类
directory	仅用于 SimplePropertiesWriter。表示属性文件的目录。若没有指定, 默认为 "conf"
filename	仅用于 SimplePropertiesWriter。属性文件的名称。若没有指定, 默认为 requestHandler 的名称 (根据 solrconfig.xml 中的定义, 后缀 .properties, 即 dataimport.properties)
locale	地区。若没有定义, 则使用 ROOT 地区。它必须是语言 - 国家这种格式, 如 en-US, zh-CN

#### 5. DataSource 数据源

数据源是对 Solr 索引数据的外部来源进行抽象, 你可以通过编写继承了 org.apache.solr.handler.dataimport.DataSource 类来创建自定义数据源。

数据源定义的必需属性为其名称和类型。名称标识了实体元素对应哪个数据源。

可用的数据源类型如下:

##### (1) ContentStreamDataSource

它接收 POST 数据作为数据源。它可以在任何使用了 DataSource<Reader> 的 EntityProcessor 上使用。

##### (2) FieldReaderDataSource

它可用于数据库字段包含了你希望使用 XPathEntityProcessor 来处理的 XML 内容时使用。你可以在配置中同时设置 JDBC 和 FieldReader 数据源, 如下:

```
<dataSourcename="a1" driver="org.hsqldb.jdbcDriver" ..../>
<dataSourcename="a2" type="FieldReaderDataSource" />
</document>
```

```

<entity name="e1" dataSource="a1" processor="SqlEntityProcessor" pk="docid"
  query="select * from t1 ...">
<entity name="e2" dataSource="a2" processor="XPathEntityProcessor"
  dataField="e1.fieldToUseForXPath">
  ...
</entity>
</entity>

```

FieldReaderDataSource 可以接收一个 encoding 参数，它默认为 "UTF-8"。它必须是语言 - 国家格式，如 en-US。

### (3) JdbcDataSource

默认的数据源，一般配合 SqlEntityProcessor 一起使用。

该数据源常用于 XPathEntityProcessor 来从一个底层的 file:// 或 http:// 地址获取内容。示例如下：

```

<dataSource name="a"
  type="URLDataSource"
  baseUrl="http://host:port/"
  encoding="UTF-8"
  connectionTimeout="5000"
  readTimeout="10000"/>

```

URLDataSource 类型接收可选参数如表 2-6 所示。

表 2-6 URLDataSource 类型接收可选参数

可选参数	描 述
baseUrl	指定路径名称的 baseUrl。你可以使用它在 Develop/QA/Produce 环境间指定 host/port 进行切换。使用这个属性使得不需要改变 solrconfig.xml 文件
connectionTimeout	指定连接超时时间的毫秒值。默认为 5000ms
encoding	响应头中使用的默认编码。可以用它来覆盖默认编码。默认编码为 UTF-8
readTimeout	指定读操作的超时时间毫秒值。默认为 10000ms

## 6. EntityProcessor 实体处理器

实体处理器提取、转换数据并将其添加到 Solr 索引中。实体一般可以是数据库中的视图或表。

每个处理器有自己的属性集合，任何实体都通用的属性如表 2-7 所示。

表 2-7 实体通用属性

通用属性	描 述
dataSource	数据源名称。可以定义多个数据源，使用该属性来给实体配置数据源名称
name	必需参数。用于标识实体的唯一名称
pk	实体主键。可选参数，仅在需要使用 delta-import 时需要。它和 schema.xml 中定义的 uniqueKey 没有关系，但可以是相同的。如果你要做 delta-import 时是必需的，且可以以 \${dataimporter.delta.<column-name>} 引用字段名作为主键

(续)

通用属性	描 述
processor	若没有指定则默认值为 SqlEntityProcessor。只有当 dataSource 不是 RDBMS 类型时才需要指定此属性
onError	可选值有 abort/skip/continue。默认值为 abort。'Skip' 表示跳过当前 Document 不处理。'Continue' 表示忽略异常继续处理
preImportDeleteQuery	在执行 full-import 命令之前, 使用这个查询来清空索引数据, 替代使用 delete-ByQuery(*:*) 方式清空索引, 一般用于子 Document 的 Entity 中
postImportDeleteQuery	与上面属性类型, 唯一区别就是它是在 full-import 完成后执行
rootEntity	默认 Document 元素在的 Entity 就是 rootEntity, 如果 rootEntity 设置为 false, root-Entity 下的 Entity 会被归入 rootEntity, 最终 rootEntity 返回的每一行都会被解析为 Solr 里的一个 Document
transformer	可选的, Entity 被映射成多个 Row 后可以配置一个或多个 transformers 来对每个 Row 的数据进行清洗、筛选、转换
cacheImpl	可选的, 用于指定缓存实现类, 该类必须实现 DIHCache。该 cache 用于缓存 Entity 对象。默认实现是 SortedMapBackedCache
cacheKey	如果指定了 cacheImpl 属性, 则需要指定需要缓存的 Entity 的 name 作为缓存主键
cacheLookup	如果指定了 cacheImpl 属性, 需要使用 Entity+name 在 cache 中查找 Entity 实例

DIH 提供 Entity 的缓存以避免对相同名称 Entity 的重复创建, 充分利用缓存提升查询的性能。默认的 SortedMapBackedCache 是一个 HashMap, 其 Key 是 Row 中的一个字段, 而其 Value 是具有相同 Key 的一系列 Row。

下面示例中, 每个 manufacturer 实体都使用 id 属性作为缓存键进行了缓存。对每个 product 实体的基于产品的 manu 属性的查询将直接查看缓存。当缓存在特定 Key 上没有数据时, 查询会运行并填充缓存。

```
<entityname="product" query="select description,sku, manu from product" >
<entityname="manufacturer" query="select id, name from manufacturer"
cacheKey="id" cacheLookup="product.manu" cacheImpl="SortedMapBackedCache"/>
</entity>
```

### (1) SqlEntityProcessor

SqlEntityProcessor 是默认的处理程序, 其关联的数据源应该为一个 JDBC URL。该处理器特有的实体属性如表 2-8 所示。

表 2-8 SqlEntityProcessor 处理器的实体属性

属性	描 述
query	必需属性。指定一条 SQL 语句来返回表中的多条 Rows
deltaQuery	增量导入时需要指定的 SQL 语句, 此 SQL 语句会作为增量更新的一部分并返回只包含主键 PK 的结果集 Rows, 并且返回的主键 PK 可以通过 \${dataimporter.delta.<column-name>} 变量在 deltaImportQuery 中被引用
parentDeltaQuery	增量导入时需要指定的 SQL 语句

(续)

属性	描 述
deletedPkQuery	增量导入时需要指定的 SQL 语句
deltaImportQuery	增量导入时需要指定的 SQL 语句, 如果这个 SQL 语句没有指定, 那么 DIH 会尝试通过修改 query 属性来构造 SQL 语句, 此时 <code>\${dataimporter.delta.&lt;column-name&gt;}</code> 变量会被使用。比如: <code>select * from tbl where id=\${dataimporter.delta.id}</code>

## (2) XPathEntityProcessor

该处理器用于索引 XML 格式数据, 其数据源通常是 `URLDataSource` 或 `FileDataSource`。Xpath 也可用于下面的 `FileListEntityProcessor`, 来从每个文件中生成索引文档。

该处理器特有的实体属性如表 2-9 所示。

表 2-9 XPathEntityProcessor 处理器特有的实体属性

属性	描 述
Processor	必需属性且必需设置为 <code>XpathEntityProcessor</code>
url	必需属性, 值为一个 Http 链接或者文件路径
stream	可选属性: 对于文件下载或大文件, 需要将其设置为 <code>true</code>
forEach	当你没有指定 <code>useSolrAddSchema</code> 属性时需要此属性。使用此属性来建立 record 处理循环, 每个循环内可以使用 XPath 表达式来提取想要的数
xsl	可选属性, 值可以为 xsl 文件的 http 链接或者文件系统路径。Xsl 文件转换功能可以作为 XML 文件的预处理器
useSolrAddSchema	如果提供的 XML 文件内容是 Solr 更新 XML 要求的规范格式, 那么请将此属性设置为 <code>true</code>
flatten	可选属性: 如果设置为 <code>true</code> , XML 文件中的所有标签都将被当作一个字符串提取出来存入一个域中

实体中的每个字段元素都可以有的属性如表 2-10 所示。

表 2-10 字段元素属性

属性	描 述
xpath	必需属性, 使用 XPath 表达式为当前域从当前 record 中提取内容。只支持部分 XPath 表达式语法
commonField	可选属性, 若设置为 <code>true</code> , 则表明当前域是一个通用域, XPath 表达式解析出来的内容会被复制到其他 document 中

## (3) MailEntityProcessor

`MailEntityProcessor` 使用 Java Mail API 基于 IMAP 协议索引 email 消息。

`MailEntityProcessor` 通过使用用户名密码来连接到特定的邮箱, 获取每个消息的 email 头部, 然后获取完整的 email 内容来构造一个索引文档 (每个邮件对应一个索引文档), 使用示例如下列代码所示:

```
<dataConfig>
<document>
```



```

<entity processor="MailEntityProcessor"
  user="email@gmail.com"
  password="password"
  host="imap.gmail.com"
  protocol="imaps"
  fetchMailsSince="2009-09-20 00:00:00"
    batchSize="20"
  folders="inbox"
    processAttachement="false"
  name="sample_entity"/>
</document>
</dataConfig>

```

MailEntityProcessor 特有的实体属性如表 2-11 所示。

表 2-11 MailEntity Processor 处理器特有的实体属性

属性	描 述
processor	必需属性且属性值必须为 MailEntityProcessor
user	必需属性。表示用于登录认证 IMAP server 的用户名
password	必需属性。表示用于登录认证 IMAP server 的密码
host	必需属性。表示 IMAP server 的主机名
protocol	必需属性。表示使用的 IMAP 协议。可选值有：imap, imaps, gimap, and gimaps
fetchMailsSince	可选属性。指定日期或时间之前的邮件将会被过滤掉不被导入。期望的日期格式为：yyyy-MM-dd HH:mm:ss
folders	必需属性。表示从哪些文件夹里拉取邮件消息，比如 inbox，多个值用逗号分隔
recurse	可选属性（默认为 true）。用于指示 Processor 是否需要递归导入所有子文件夹下的邮件消息
include	可选属性。表示哪些文件夹下的邮件消息需要被处理导入，支持正则表达式，多个值用逗号分隔
exclude	可选属性。与上面的属性类型，这里表示哪些文件夹下的邮件消息需要排除掉不处理
processAttachement/ processAttachements	可选属性（默认为 true）。设置 true 表示使用 Tika 来处理邮件里的附件
includeContent	可选属性（默认为 true）。是否需要 Message Body 进行处理，并构造 Solr 的 Document

在执行 full-import 之后，MailEntityProcessor 处理器会记录上一次导入的时间戳，以便随后的导入操作可以使用 fetchMailsSince 过滤器保证仅仅只拉取邮件服务器里最新的邮件。这个行为是自动发生的。举例说明，假如你在 mail-data-config.xml 中将 fetchMailsSince 设置为 2014-08-22 00:00:00，然后在指定日期之后的所有邮件信息在第一次执行导入操作时将会被导入，随后的导入操作将会将上一次导入操作时间作为 fetchMailsSince 的属性值来过滤出最新的邮件消息。这样就保证了每次只有自上一次导入之后产生的新邮件才会被导入。

当连接到 Gmail 账户时，你可以通过设置协议为 gimap 或 gimaps 来提升 MailEntity-Processor 的处理性能。它允许 processor 将 fetchMailsSince 发送到 Gmail server，然后 Gmail server 在服务器端使用日期过滤器对返回的邮件消息进行过滤，然而，Gmail 只支持到 date



精度，所以如果一天运行超过一次，那么服务器端的过滤器可能会返回之前已阅读过的旧邮件。

#### (4) TikaEntityProcessor

TikaEntityProcessor 使用 Apache Tika 来处理传入的文档，配置示例如下：

```
<dataConfig>
<dataSource type="BinFileDataSource" />
<document>
<entity name="tika-test" processor="TikaEntityProcessor"
url="../contrib/extraction/src/test-files/extraction/solr-word.pdf"
format="text">
<field column="Author" name="author" meta="true"/>
<field column="title" name="title" meta="true"/>
<field column="text" name="text"/>
</entity>
</document>
</dataConfig>
```

该处理器特有的实体属性如表 2-12 所示。

表 2-12 TikaEntityProcessor 处理器特有的实体属性

属性	描 述
dataSource	此属性用于定义数据源后续的配置中可以使用 name 属性引用该数据源 Processor 的可用的数据源类型如下： <input type="checkbox"/> BinURLDataSource：用于 HTTP 资源或者磁盘文件 <input type="checkbox"/> BinContentStreamDataSource：用于通过文件流上传的文件 <input type="checkbox"/> BinFileDataSource：用于本地文件系统里的文件
url	源文件的网络链接或硬盘路径，必需属性
htmlMapper	此属性用于控制 Tika 如何解析 HTML。“default” mapper 会剔除 document 的所有 html 元素，而“identity” mapper 会保留所有 html 元素不做任何修饰。若需要指定当前属性，那么属性值必须是 default 或 identity；若当前属性未指定，默认值为 default
format	指定输出格式，可选值有 text、xml、html、none 若未指定的话，默认值为 text。如果只索引 metadata 元数据信息不索引 documentde body，那么 format 设置为 none
parser	默认的 parser 类为 org.apache.tika.parser.AutoDetectParser。若你需要指定为其他 parser 实现类或者自定义的 parser 实现类，请输入该类的完整包路径
fields	用于指定从输入的 Document 中提取的 field 列表以及它们是如何映射至 Solr 的 Field 的。若 meta 属性设置为 true，那么 field 将直接从 Document 的 metadata 元数据信息里获取，而不是从 document body 的主文本里解析。
extractEmbedded	若设置为 true 即表示命令 TikaEntityProcessor 提取内嵌的 Document 或附件，否则直接忽略
onError	当发生异常时，默认 TikaEntityProcessor 会停止 Document 处理操作，当操作失败时，Tika-EntityProcessor 会跳过该 Document 不做处理

#### (5) FileListEntityProcessor

该处理器基本就是一个封装器，它设计用于生成一系列满足由属性指定的条件的文件，然后这些文件可以被传递给另一个处理器，比如 XPathEntityProcessor。该处理器的实体信

息将内嵌在 FileListEntity 实体中。它会生成四个隐式的字段：fileAbsolutePath、fileSize、fileLastModified、fileName，它们可用在内嵌的处理器中。该处理器没有使用数据源，该处理器特有的属性如表 2-13 所示。

表 2-13 FileListEntityProcessor 处理器特有的属性

属性	描 述
fileName	必需属性。使用一个正则表达式标识所有匹配的文件
basedir	必需属性。设置基准目录（绝对路径）
recursive	是否递归搜索目录下的文件，默认为 false
excludes	一个正则表达式用来标识不被包含进来的文件
newerThan	表示一个 yyyy-MM-ddHH:mm:ss 格式的日期时间或者 NOW - 2YEARS 这种表达式表示的日期，只有在指定日期之前的文件才会被包含
olderThan	与 newerThan 类型，olderThan 表示指定日期之后的文件才会被包含
rootEntity	这个属性应该设置为 false，这样可以确保每一行（即文件路径）会被当作一个文档
dataSource	必须设置为 null

下面的示例组合了 FileListEntityProcessor 和另一个会针对每个找到的文件生成一系列字段的处理器。

```
<dataConfig>
<dataSource type="FileDataSource"/>
<document>
<!-- this outer processor generates a list of files satisfying the conditions
      specified in the attributes -->
<entity name="f" processor="FileListEntityProcessor"
  fileName="*.xml"
  newerThan="'NOW-30DAYS'"
  recursive="true"
  rootEntity="false"
  dataSource="null"
  baseDir="/my/document/directory">
<!-- this processor extracts content using Xpath from each file found -->
<entity name="nested" processor="XPathEntityProcessor"
  forEach="/rootelement" url="{f.fileAbsolutePath}" >
<field column="name" xpath="/rootelement/name"/>
<field column="number" xpath="/rootelement/number"/>
</entity>
</entity>
</document>
</dataConfig>
```

#### (6) LineEntityProcessor

LineEntityProcessor 会依次逐行读取数据源中的内容，并对每个读取到的行返回名为 rawLine 的字段，其内容不会被解析；但是可以添加转换器来操作 rawLine 字段的数据，或创建其他额外的字段。

读取的行数据可由两个属性 `acceptLineRegex` 和 `omitLineRegex` 指定的正则表达式进行过滤，如表 2-14 所示。

表 2-14 LineEntityProcessor 处理器的属性

属性	描 述
url	一个必需的属性，指定输入文件的位置且需要兼容配置的数据源。如果这个值是相对路径且你使用的是 <code>FileDataSource</code> 或 <code>URLDataSource</code> ，那么此属性会被判定为相对路径
acceptLineRegex	可选属性，不匹配指定正则表达式的行将会被丢弃
omitLineRegex	可选属性，匹配指定正则表达式的行将会被忽略不处理

### (7) PlainTextEntityProcessor

`PlainTextEntityProcessor` 读取数据源中的所有内容作为单个隐式字段 `plainText`，其内容不会被解析；然而你可以添加转换器来操作 `plainText` 字段的数据，或创建其他额外的字段，例如：

```
<entityprocessor="PlainTextEntityProcessor" name="x" url="http:// abc.com/a.txt"
dataSource="data-source-name">
<!-- copies the text to a field called 'text' in Solr-->
<fieldcolumn="plainText" name="text" />
</entity>
```


使用 `PlainTextEntityProcessor` 处理器请确保数据源是 `DataSource<Reader>CFileDataSource`、`URLDataSource`) 类型的。

### (8) 如何自定义 EntityProcessor

当 Solr 内置提供的这些 `EntityProcessor` 仍然无法满足你的需求时，你可以通过继承 `EntityProcessor` 抽象类或者继承 `EntityProcessorBase` 类来实现自定义 `EntityProcessor`，其实 `EntityProcessorBase` 就是 `EntityProcessor` 抽象类的一个子类，这里推荐你继承 `EntityProcessorBase` 类。继承 `EntityProcessorBase` 类后，你需要重写其 `init (Context context)`、`nextRow()`、`getNext()`、`destroy()` 这几个主要函数，具体如何实现，可以参阅 Solr 内置的 `LineEntityProcessor` 实现类源码。

自定义 `EntityProcessor` 代码编写完毕后，你需要将其打包成 jar，然后将 jar 包放置到 core 的 lib 目录下，之后就在 `data-config.xml` 里应用你自定义的 `EntityProcessor` 啦，具体配置示例如下：

```
<dataSource type="FileDataSource" />
<document>
<entity name="f" processor="foo.MyEntityProcessor" .../>
... ..//
</document>
```

 **注意** `processor` 属性值为自定义的 `EntityProcessor` 类时，需要输入自定义 `EntityProcessor` 类的完整包路径。

## 7. transformer 转换器

转换器可以操作由实体返回的文档中字段。一个转换器可以创建新的字段或者修改已有的字段。你需要通过在 `<entity>` 元素上添加 `transformer` 属性来告诉实体你的导入操作将使用哪些转换器。`transformer` 属性可以指定多个，多个属性值使用逗号分割，示例如下：

```
<entityname="abcde" transformer="ClobTransformer,DateFormatTransformer" />
```

除开特殊情况外，`transformer` 属性值必须包含完整包路径，除非类的包名是 `org.apache.solr.handler.dataimport`，那么包名可以省略直接输入类名即可，或者如果该类属于 Solr 源码包中的类，那么属性值等于 `solr.<classname>` 也是可行的。这条规则同样适用于所有 Solr 插件比如 `DataSource`、`EntityProcessor`、`Evaluator`。

多个 `transformer` 转换器严格按照指定的顺序从左至右依次被执行，`transformer` 转换器的一些转换规则参数通过为随后的 `field` 元素添加属性方式来指定，比如：

```
<entityname="e" transformer="ClobTransformer" ...>
<fieldcolumn="hugeTextField" clob="true" />
...
</entity>
```

Solr 提供了如表 2-15 所示的这些内置 `transformer` 转换器：

表 2-15 内置 transformer 转换器

名称	描 述
ClobTransformer	用于将关系型数据库里的 Clob 类型转换成 String
DateFormatTransformer	用于将日期时间格式字符串类型转换成 <code>java.util.Date</code> 对象
HTMLStripTransformer	用于剔除 HTML 标签
LogTransformer	用于记录日志数据，可以记录到日志文件或控制台
NumberFormatTransformer	用于将一个字符串转换成数字，内部使用 Java 的 <code>NumberFormat</code> 类实现
RegexTransformer	使用正则表达式对 Field 的值进行再次转换
ScriptTransformer	使用 JavaScript 脚本或者其他 Java 支持的脚本语言来编写转换函数来实现 Field 值的转换
TemplateTransformer	使用模板来转换 Field 值

### (1) RegexTransformer


Solr DIH 内置提供了一个 `RegexTransformer` 转换器，它使用正则表达式来提取或操作 Field 的值。它的实际实现类为 `org.apache.solr.handler.dataimport.RegexTransformer`。由于它属于默认包路径下，故配置时包名可以省略。

该转换器包含如下可配置属性：

- `regex`：指定应用到 `column` 或 `sourceColName` 上的正则表达式。若 `replaceWith` 属性不存在，那么会提取每个正则匹配组的值并以 `list` 集合形式返回；

- ❑ **sourceColName** : 正则表达式应用到的源列名, 如果此属性未指定, 那么源列和目标列将是同一个, 即直接替换更新源列的值, 而不是存储到一个新的目标列上;
- ❑ **splitBy**: 用于对一个列的值按照分隔符进行分割返回一个 lis;
- ❑ **groupNames**: 此属性用于接收以逗号分割的多个 column 名称, 它依次对应正则表达式的每个匹配组, 匹配组提取到的值会作为对应 column 的值, 如果某些组没有对应的 column, 则会使用空格代替;
- ❑ **replaceWith**: 使用正则表达式对 column 的值进行替换, 它等价于 Java 里的 `new String(<sourceColVal>).replaceAll(<regex>, <replaceWith>)`。

---

 **注意** 只有当 field 配置了 `regex` 或 `splitBy` 属性时, `RegexTransformer` 转换器才会被激活启用。

---

下面是一个 `RegexTransformer` 的简单使用示例:

```
<entity name="foo" transformer="RegexTransformer"
query="select full_name , emailids from foo"/>
<field column="full_name"/>
<field column="firstName" regex="Mr(\w*)\b.*" sourceColName="full_name"/>
<field column="lastName" regex="Mr.*?\b(\w*)" sourceColName="full_name"/>
<!-- another way of doing the same -->
<field column="fullName" regex="Mr(\w*)\b(.*)" groupNames="firstName,lastName"/>
<field column="mailId" splitBy="," sourceColName="emailids"/>
</entity>
```

## (2) ScriptTransformer

你可以使用 JavaScript 或者其他 Java 支持的脚本语言来编写 transformers, 要使用这项功能前提是你必须使用 JDK 6 或更高的版本。

```
<dataConfig>
<script><![CDATA[
    function f1(row)
    {
        row.put('message', 'Hello World!');
        return row;
    }
    ]]>
</script>
<document>
<entity name="e" pk="id" transformer="script:f1" query="select * from X">
    ....
</entity>
</document>
</dataConfig>
```

另外一个稍微复杂点的例子:

```

<dataConfig>
<script><![CDATA[
    function CategoryPieces(row)    {
        var pieces = row.get('category').split('/');
        var arr = new java.util.ArrayList();
        for (var i=0; i<pieces.length; i++) {
            arr.add(pieces[i]);
        }
        row.put('categorypieces', arr);
        row.remove('category');
        return row;
    }
    ]]>
</script>
<document>
<entity name="e" pk="id" transformer="script:CategoryPieces" query="select * from X">
    ....
</entity>
</document>
</dataConfig>

```

- ❑ 你可以在 dataConfig 元素内部添加一个 script 标签，默认会被解析为 Javascript。如果你需要使用其他脚本语言，你需要在 script 标签里通过 language 属性来指定，前提是 Java 6 支持该脚本语言
- ❑ 你可以使用脚本编写任何你想要实现的转换函数，每个转换函数需要强制接收一个 row 变量，该变量是一个类似 Map<String, Object> 的结构，在函数转换完成后你需要再返回一个 row 类型。
- ❑ 要移除某个 Entity，你可以在转换函数内部通过 row.remove(keyname) 实现
- ❑ 想要为某个 field 添加多个 Entity，你必须使用 var arr = new java.util.ArrayList()，不能使用 JavaScript 里的 Array 数组类型。
- ❑ 使用脚本转换函数来创建一个 Entity，你可以通过在 Entity 节点元素中指定 transformer="script:<function-name>" 来实现。
- ❑ 在上面的 data-config 配置中，query 属性指定的 SQL 语句执行后返回的结果集中的每个 Row 都会经过 CategoryPieces 函数进行转换，转换后生成的每个新 Entity 缓存到 Entity="e" 上。
- ❑ 脚本实现的转换函数在执行语义上讲，跟 Java 实现的 transformer 相同，Java 里该转换函数定义为 transformRow(Map<String, Object>, Context context)，其实你会第二个参数 context 即执行上下文对象，在脚本实现里第二个参数虽然被忽略了，但并不影响执行。

### (3) DateFormatTransformer

此转换器用于将日期时间格式字符串类型转换成 java.util.Date 对象，当且仅当 field 元



素添加了 `dateTimeFormat` 属性时, `DateFormatTransformer` 转换器才会被激活启用。使用示例如下:

```
<field column="date" xpath="/RDF/item/date"
dateTimeFormat="yyyy-MM-dd'T'HH:mm:ss" locale="en" />
```

此转换器包含如下可配置属性如表 2-16 所示。

表 2-16 `DateFormatTransformer` 转换器的可配置属性

名称	描 述
<code>dateTimeFormat</code>	用于指定需要转换的日期时间字符串的格式, 它完全遵循 Java 里的 <code>SimpleDateFormat</code> 类的日期格式规范
<code>sourceColName</code>	<code>dateTimeFormat</code> 参数将被应用到 <code>sourceColName</code> 属性指定的列上, 若此属性未指定, 那么源列和目标列将是同一个, 即转换函数会替换更新源列的值
<code>Locale</code>	用于指定本地所属地区, 若未指定, 对于 Solr 4.1 版本或者更高版本会使用 <code>ROOT</code> local 作为默认值, Solr 4.1 版本之前会使用当前机器的默认 <code>Local</code>

#### (4) `NumberFormatTransformer`

此转换器用于将一个字符串转换成数字, 配置示例如下:

```
<field column="price" formatStyle="number" />
```

此转换器包含如下可配置属性如表 2-17 所示。

表 2-17 `NumberFormatTransformer` 转换器的可配置属性

名称	描 述
<code>formatStyle</code>	用于指定转换字符串的数字格式, 该属性值必须是 ( <code>number percent integer currency</code> ) 其中之一, 它完全遵循 Java 里的 <code>NumberFormat</code> 类的数字格式规范
<code>sourceColName</code>	<code>formatStyle</code> 参数将被应用到 <code>sourceColName</code> 属性指定的列上, 若此属性未指定, 那么源列和目标列将是同一个, 即转换函数会替换更新源列的值
<code>Locale</code>	用于指定本地所属地区, 若未指定, 对于 Solr 4.1 版本或者更高版本会使用 <code>ROOT</code> local 作为默认值, Solr 4.1 版本之前会使用当前机器的默认 <code>Local</code>

#### (5) `TemplateTransformer`

此转换器用于根据一个静态的字符串模板来生成一个新的 field 或者更新一个 field 的值, 配置示例如下:

```
<entity name="e" transformer="TemplateTransformer" ..>
<field column="namedesc" template="hello${e.name},${e.parent.surname}" />
</entity>
```

此转换器包含的可配置属性如表 2-18 所示。

表 2-18 TemplateTransformer 的可配置属性

名称	描述
template	用于指定模板字符串, 上述例子中使用了 '\${e.name}' 和 '\${e.parent.surname}' 两个占位符参数。

### (6) HTMLStripTransformer

此转换器用于去除 HTML 标签只保留下纯文本, 当你在处理 HTML 文件内容时, 可能只关心标签之间的文本内容, 此时 HTMLStripTransformer 就派上用场啦。此转换器内部是通过 org.apache.solr.analysis.HTMLStripReader 类来实现该功能的。配置示例如下:

```
<entity name="e" transformer="HTMLStripTransformer" ...>
<field column="htmlText" stripHTML="true" />
...
</entity>
```

此转换器包含的可配置属性如表 2-19 所示。

表 2-19 HTMLStripTransformer 转换器的可配置属性

名称	描述
stripHTML	Boolean 值, 设置为 true 即表示启用 HTMLStripTransformer 功能

### (7) ClobTransformer

此转换器用于处理将关系型数据库里的 Clob 类型数据转换成 String。当你读取数据库表数据时, 若某个字段的类型为 Clob 类型, 此时 ClobTransformer 转换器就显得很有用了。配置示例如下:

```
<entity name="e" transformer="ClobTransformer" ...>
<field column="hugeTextField" clob="true" />
...
</entity>
```

此转换器包含的可配置属性如表 2-20 所示。

表 2-20 ClobTransformer 转换器的可配置属性

名称	描述
clob	Boolean 值, 设置为 true 即表示启用 ClobTransformer 功能
sourceColName	表示 ClobTransformer 处理的源列名, 若此属性未指定, 那么源列和目标列将是同一个, 即转换函数会替换更新源列的值

### (8) LogTransformer

此转换器用于记录日志数据, 可以将打印到控制台或者写入日志文件中。配置示例如下:

```
<entity ...
transformer="LogTransformer"
logTemplate="The name is ${e.name}" logLevel="debug" >
```

```
....
</entity>
```

与其他转换器不同的是，此转换器不需要作用于任何 field 上，所以它的属性是直接配置在 entity 上的。

可选的日志级别如下：

- ☐ trace;
- ☐ debug;
- ☐ info;
- ☐ warn;
- ☐ error。

---

 **注意** 上面的日志级别参数值是大小写敏感的，必须全部小写。

---

### (9) 如何自定义 Transformer 转换器

如果 Solr 内置提供的这些转换器仍然无法满足你的需求时，你可以自己自定义 Transformer 转换器。

举个例子，假设你的 schema 中有一个 string 类型的单值域 artistName，域值可以包含多个单词比如 "Celine Dion"，但是这里会有一个问题，你的数据开头和结尾可能会包含空格字符，由于这里使用的是 string 类型，所以 solr 提供的 WhitespaceAnalyzer 分词器将无法使用，一种解决方法就是使用自定义 Transformer 转换器实现 TrimTransformer 来完成头尾空格字符的去除。

```
package foo;

public class TrimTransformer {
    public Object transformRow(Map<String, Object> row) {
        String artist = row.get("artist");
        if (artist != null)
            row.put("ar", artist.trim());
        return row;
    }
}
```

你不需要继承任何类，你只需要简单编写一个普通类，然后在该类内部定一个 transformRow 函数，该函数声明必须如上述代码所示。

当然你也可以继承抽象类 org.apache.solr.handler.dataimport.Transformer 来实现。

编写完毕后你需要将其打包成 jar，将 jar 包复制到 Core 的 lib 目录下，然后就可以在 data-config.xml 配置文件中引用你自定义的 transformer 转换器了，配置示例如下所示：

```
<entity name="artist" query="..." transformer="foo.TrimTransformer">
```

```
<field column="artistName" />
</entity>
```

在上述自定义 Transformer 示例代码中，我们的 column 列名是硬编码在代码中的，从而导致这个自定义 Transformer 转换器无法达到通用效果。此时你需要在 data-config.xml 的 field 元素上自定义一个标志位属性，然后需要继承 org.apache.solr.handler.dataimport.Transformers 抽象类，重写其 transformRow 函数，根据函数的第二个参数 context 对象，你可以获取到 Entity 下声明的所有 field，通过自定义的标志位属性来动态定位 field，从而解决了 column 名称硬编码问题。具体实现代码如下所示：

```
package foo;

public class TrimTransformer extends Transformer {
    public Map<String, Object> transformRow(Map<String, Object> row, Context context) {
        List<Map<String, String>> fields = context.getAllEntityFields();
        for (Map<String, String> field : fields) {
            // 检查 data-config.xml 里的 field 元素上是否有声明 trim 属性
            String trim = field.get("trim");
            if ("true".equals(trim)) {
                // 获取当前列名
                String columnName = field.get(DataImporter.COLUMN);
                // 获取当前列的值
                Object value = row.get(columnName);
                // 去空格并将更新后的值再赋值给当前行
                if (value != null)
                    row.put(columnName, value.toString().trim());
            }
        }
        return row;
    }
}
```



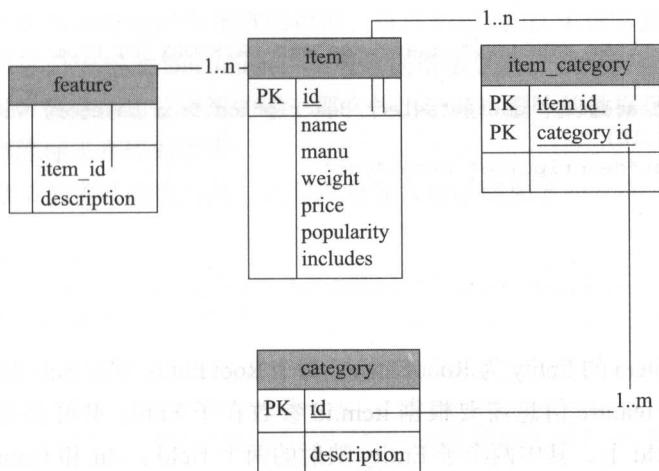
**注意** 如果该域是一个多值域，那么 Object value = row.get(columnName); 这句返回的将会是一个 java.util.List 类型。

想要继承 transformer 抽象类，你需要添加 apache-solr-dataimporthandler.jar 依赖。

## 2.3 Solr Full Import 全量导入

所谓全量索引一般指的是每次从数据库中读取需要导入的全部数据，然后提交到 Solr Server，最后删除指定 Core 的所有索引数据进行重建。全量导入一般在数据首次导入或者备份数据恢复时执行。

下面以从 MySQL 数据库全量导入多个关联表数据到 Solr 为例进行讲解说明。假设在我们的数据库中有下面几张表：



我们可能希望将 item 表的所有字段以及 item 的 category 信息、item 的 description 描述信息一并导入到 Solr 的指定 Core 中，因此 Solr 指定 Core 的 schem.xml 中可能需要预先定义好如下几个 field: name、manu、weight、price、popularity、includes、category、description。首先能想到的一个比较简单的方式就是直接使用 SQL 的多表连接返回 Solr 需要的那些 field 数据，SQL 语句类似这样：

```

select i.id,i.name,i.manu,i.weight,i.price,i.popularity,i.includes,c.description
as category,f.description
from item i,item_category ic,category c, feature f
where i.id=ic.item_id and ic.Category_id = c.id and i.id = f.item_id
  
```

这种导入方式前面章节已经做过说明，这里介绍另一种方式，其实 data-config.xml 里的 entity 元素是可以嵌套的，具体配置如下：

```

<dataConfig>
<dataSource driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/
test" user="root" password="123"/>
<document name="products">
<entity name="item" query="select * from item">
<field column="ID" name="id" />
<field column="NAME" name="name" />
<field column="MANU" name="manu" />
<field column="WEIGHT" name="weight" />
<field column="PRICE" name="price" />
<field column="POPULARITY" name="popularity" />
<field column="INSTOCK" name="inStock" />
<field column="INCLUDES" name="includes" />
<entity name="feature" query="select description from feature where item_id=
'${item.ID}'">
<field name="features" column="description" />
  
```



```

</entity>
<entity name="item_category" query="select CATEGORY_ID from item_category where
item_id='${item.ID}'">
  <entity name="category" query="select description from category where id = '${item_
category.CATEGORY_ID}'">
    <field column="description" name="cat" />
  </entity>
</entity>
</entity>
</document>
</dataConfig>

```

这里的名称为 item 的 Entity 为 Root Entity，每个 Root Entity 对应 Solr 里的一个 Document，但 category 信息和 feature 信息需要根据 item.id 参数在子 Entity 里再去关联其他表查询返回，最后映射到 field 上，其中两个子 Entity 映射的两个 field：cat 和 features 最终都会纳入 Root Entity 对应的 Document 的域中。

Solr 全量导入接口 URL

```
http://ip:port/solr/core_name/dataimport?command=full-import
```

## 2.4 Solr Delta-import 增量导入

当索引数据量很大时，每次都依靠全量导入数据显然很不切实际，所以增量导入索引数据显得格外重要。

增量导入操作内部是新开辟一个新线程来完成，并且此时 Core 的 dataimport 运行状态为 status = "busy"。增量导入耗时时间取决于需要增量导入的数据集合大小。任何时刻，你都可以通过

```
http://localhost:8983/solr/dataimport
```

这个链接来获取到增量导入操作的运行状态。当增量导入操作被执行，它会读取存储在 conf/ deltaimport.properties 配置文件，利用配置文件里记录的上一次操作时间来运行增量查询，增量导入完成后，会更新 conf/ deltaimport.properties 配置文件里的上一次操作时间戳。首次执行增量导入时，若 conf/ deltaimport.properties 配置文件不存在，会自动新建。

```

#Mon Jun 01 10:54:27 GMT+08:00 2015
mobile.last_index_time=2015-06-01 10\:54\:26
last_index_time=2015-06-01 10\:54\:26

```

如果要使用增量导入，前提你的表必须有两个字段：一个是删除标志字段即逻辑删除标志：isdeleted，另一个则是数据创建时间字段：create\_date，字段名称不一定非得是 isdeleted 和 create\_date，但必须要包含两个表示该含义的字段。根据数据创建时间跟上一次增量导入操作时间一比对，就可以通过 SQL 语句查询出需要增量导入的数据，根据

isdeleted 字段可以查询出被标记为删除的数据，这些数据 ID 主键需要传递给 Solr，这样 Solr 就能同步删除索引中相关 Document，实现数据增量更新。如果你数据表里的数据都是物理删除，没有逻辑删除标志位字段的话，那么找出已经被删除的数据显得比较困难，所以这就是需要逻辑删除标志字段的原因。

还是以上一章节的那几张表为例，实现增量导入的示例如下：

```
<dataConfig>
  <dataSource driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/
test" user="root" password="123"/>
  <document>
    <entity name="item" pk="ID" query="select * from item"
      deltaImportQuery="select * from item where ID='${dih.delta.id}'"
      deltaQuery="select id from item where last_modified > '${dih.
last_index_time}';">
      <entity name="feature" pk="ITEM_ID"
        query="select DESCRIPTION as features from FEATURE where ITEM_ID=
'${item.ID}';"
        deltaQuery="select ITEM_ID from FEATURE where last_modified >
'${dih.last_index_time}';"
        parentDeltaQuery="select ID from item where ID=${feature.
ITEM_ID}"/>
      <entity name="item_category" pk="ITEM_ID, CATEGORY_ID"
        query="select CATEGORY_ID from item_category where ITEM_ID=
'${item.ID}';"
        deltaQuery="select ITEM_ID, CATEGORY_ID from item_category
where last_modified > '${dih.last_index_time}';"
        parentDeltaQuery="select ID from item where ID=${item_category.
ITEM_ID}"/>
      <entity name="category" pk="ID"
        query="select DESCRIPTION as cat from category where ID =
'${item_category.CATEGORY_ID}';"
        deltaQuery="select ID from category where last_modified
> '${dih.last_index_time}';"
        parentDeltaQuery="select ITEM_ID, CATEGORY_ID from item_
category where CATEGORY_ID=${category.ID}"/>
    </entity>
  </entity>
</document>
</dataConfig>
```

要理解上面的 data-config.xml 增量导入配置，你需要先理解如下几个增量导入参数：

- pk：表示当前 Entity 表示的主键字段名称，这里的主键指的是数据库表中的主键，而非 Solr 中的 UniqueKey 主键域。如果你的 SQL 语句中使用了 as 关键字为主键字段定义了别名，那么这里的 pk 属性需要相应的修改为主键字段的别名，切记；
- query：用于指定全量导入时需要的 SQL 语句，比如 select \* from xxxx where isdeleted=0，查询返回的是未被删除的所有有效数据，这个 query 参数只对全量导入有效，对增量

导入无效；

- ❑ `deltaQuery`：查询需要增量导入的记录的主键 ID 所需的 SQL 语句。可能是 update, insert, delete 等操作，比如：

```
deltaQuery="select ID from xxx where my_date > '${dataimporter.last_index_time}'"
```

此参数值对增量导入有效；

- ❑ `deletedPkQuery`：查询已经被逻辑删除了的数据所需的 SQL 语句，所以这里你需要一个类似 isdeleted 的逻辑删除标志位字段。Solr 通过此参数表示的 SQL 语句执行后返回的结果集来删除索引里面对应的数据。使用示例：

```
select ID from myinfo where isdelete=1
```

此参数值对增量导入有效；

- ❑ `deltaImportQuery`="select \* from myinfo where ID='\${dataimporter.delta.ID}'"

利用 `deltaQuery` 参数返回的所有需要增量导入的数据主键 ID，遍历每个主键 ID，然后循环执行 `deltaImportQuery` 参数表示的 SQL 语句返回所有需要增量导入的数据。

其中变量 `${dataimporter.delta.ID}` 用于获取 `deltaQuery` 返回的每个主键 ID。

增量导入的接口 URL：

```
http://localhost:8080/solr/dataimport?command=delta-import
```

我们可以手动在浏览器里输入上面接口 URL 或者在 Solr Web 界面里完成增量导入操作，但现实系统中，我们的数据库数据可能时时刻刻都在更新，我们不可能每次都依靠人工去完成增量导入来实现 Solr 索引更新。此时我们需要一个定时调度器。

对于 Linux 系统而言，可以直接借助 cron tab 任务实现，编写一个 shell 脚本：

```
#!/bin/sh
curl http://localhost:8080/solr/dataimport?command=delta-import
```

然后编写 crontab

```
*/1 * * * * deltaimport.sh
```

你也可以通过 HttpClient 工具包去定时请求增量导入的接口 URL，HttpClient 如何使用大家就自行学习吧。

此外，你也可以使用 SolrJ 类库来完成与 Solr 服务之间的交互，通过 SolrJ 实现编码方式来调用增量导入接口。

```
HttpSolrClient client = new HttpSolrClient("http://localhost:8080/solr/core1");
ModifiableSolrParams params = new ModifiableSolrParams();
params.set("command", "delta-import"); // full-import 全量导入
QueryRequest request = new QueryRequest(params);
request.setPath("/dataimport");
server.request(request);
```

然后可以通过 java 里的 Timer 类或者借助 Quartz 来实现任务周期性调度，执行上述代

码即可。最后介绍下如何使用 Solr dataimport scheduler, 其实 Solr dataimport scheduler 并不包含在任何 Solr 的正式发布版中, Solr dataimport scheduler 只是社区用户的一个提议, 属于社区开发者贡献的一个功能, 并没有纳入 Solr 正式源码中, Solr 也并没有想要添加该功能的意愿, 因为当前所有的主流操作系统都支持兼容的内置任务调度, 而 Solr 也不想再重复去造轮子。想要使用 Solr dataimport scheduler 你首先需要下载所需的 jar 包:

<http://code.google.com/p/solr-data-import-scheduler/>

由于国内 Google 已经无法使用, 所以上述下载地址需要翻墙, 不过百度网盘里还是可以搜到其他网友的无私分享。若实在找不到可用的下载资源, 可从我下面分享的百度网盘链接去下载:

<http://pan.baidu.com/s/1nuT71PZ>

获取到依赖的 jar 包后, 首先将其 copy 到 core/lib 目录下, 然后在 core/conf 下添加 dataimport.properties 配置, 该配置文件可以通过解压 solr-dataimport-scheduler.jar 包后获取到, 然后需要修改 dataimport.properties 配置如下:

```
# dataimport.properties example
#
# From this example, copy everything bellow "dataimport scheduler properties" to your
# dataimport.properties file and then change params to fit your needs
#
# IMPORTANT:
# Regardless of whether you have single or multiple-core Solr,
# use dataimport.properties located in your solr.home/conf (NOT solr.home/core/conf)
# For more info and context see here:
# http://wiki.apache.org/solr/DataImportHandler#dataimport.properties_example

#Tue Jul 21 12:10:50 CEST 2010
metadataObject.last_index_time=2010-09-20 11\12\47
last_index_time=2010-09-20 11\12\47

#####
#                                     #
# dataimport scheduler properties    #
#                                     #
#####
# to sync or not to sync
# 1 - active; anything else - inactive
syncEnabled=1

# which cores to schedule
# in a multi-core environment you can decide which cores you want synchronized
# leave empty or comment it out if using single-core deployment
syncCores=coreHr,coreEn

# solr server name or IP address
# [defaults to localhost if empty]
```

```

server=localhost
# solr server port
# [defaults to 80 if empty]
port=8080

# application name/context
# [defaults to current ServletContextListener's context (app) name]
webapp=solrTest_WEB

# URL params [mandatory]
# remainder of URL
params=/select?qt=/dataimport&command=delta-import&clean=false&commit=true

# schedule interval
# number of minutes between two runs
# [defaults to 30 if empty]
interval=10

# 重建索引的时间间隔, 单位: 分钟, 默认 7200, 即 5 天;
# 为空, 为 0, 或者注释掉: 表示永不重建索引
reBuildIndexInterval=2

# 重建索引的参数
reBuildIndexParams=/dataimport?command=full-import&clean=true&commit=true
# 重建索引时间间隔的计时开始时间, 第一次真正执行的时间等于 reBuildIndexBeginTime+reBuildIndexInterval*60*1000;
# 两种格式: 2012-04-11 03:10:00 或者 03:10:00, 后一种会自动补全日期部分为服务启动时的日期
reBuildIndexBeginTime=03:10:00

```

- ❑ syncEnabled: 表示是否同步导入, 1 表示激活同步导入, 其他值表示不启用同步导入;
- ❑ syncCores: 表示需要哪些 Core 进行调度, 多个 Core 名称可以使用逗号进行分割;
- ❑ server: 表示 Solr Server 的 IP 或者域名;
- ❑ port: 表示 Solr Server 监听的端口号;
- ❑ webapp: 表示 solr web application 名称, 默认为 Solr;
- ❑ params: 表示 dataimport URL 以及需要附带的请求参数, 比如 /core/dataimport?command=delta-import&clean=false&commit=true;
- ❑ interval: 表示每间隔多少分钟调度执行一次, 默认为 30 分钟。

然后你还需要在 Solr web application 的 web.xml 里配置 ApplicationListener, 配置如下:

```

<listener>
<listener-class>
org.apache.solr.handler.dataimport.scheduler.ApplicationListener
</listener-class>
</listener>

```

然后你需要在 solrconfig.xml 中配置 DataImportHandler, 并指定 data-config.xml 文件加



载路径,紧接着需要在 data-config.xml 里配置好的你的增量导入,最后重启 Tomcat 使其能够重新部署 Solr web application,从而能够重新解析 web.xml,保证刚刚配置的 Application-Listener 能够生效。到此, Solr 的 delta import scheduler 就搞定了。

## 2.5 Solr 索引

虽说 Solr 是基于 Lucene 实现的,用户甚至不需要了解 Lucene 就可以轻松上手 Solr,但为了更好的使用 Solr,还是有必要稍微了解下 Lucene 的内部索引原理。

### 2.5.1 Lucene 索引原理

Lucene 建立索引的过程其实就是创建倒排索引表的过程,为此, Lucene 设计了一个良好的索引结构, Lucene 索引结构中有几个基本概念:索引 Index、文档 Document、词 Term、域 Field、段 Segment。

#### □ 索引 (Index):

- 在 Lucene 中一个索引是放在一个文件夹中的,一个索引就是多个 Document 的集合。
- 同一个索引目录中的所有的文件构成一个 Lucene 索引。
- 一个索引其实就是多个 Document 的集合。

#### □ 文档 (Document):

- 文档是我们构建索引的基本单位,索引中的每个 Document 就好比数据库表中的每条记录 Record,虽然不是一个概念,但你可以这样类去理解。
- 新添加的文档是单独保存在一个新生成的段文件中。

#### □ 域 (Field):

- 一个 Document 其实就是一个 Field 的集合,每个 Field 就好比数据库表中的每个字段 column。
- 不同 Field 可以分别存储不同信息,以及拥有各自不同的存储方式。

#### □ 段 (Segment):

- 当添加一个新文档就会生成一个新的段,并且也会触发段文件合并。
- 一个索引可以包含多个段文件,段与段之间是相互独立的。
- 合并段文件有助于提升创建索引的性能。
- 段文件里记录了索引中包含多少个段,每个段包含多少个文档。
- 索引可以由多个子索引构成,这个子索引便叫做段。

#### □ 词 (Term):

- 每个 Field 的域值经过分词器处理后得到的每一项称作 Term。
- 它是索引中的最小单元。

- 在两个不同 field 中的同一个字符串被认为是不同的 term。
- Field 的域值经过序列化 (tokenized) 成 Term 集合 (Terms)。

理解上述这些基本概念, 我们还需要了解下, Lucene 的倒排索引表到底怎样一个基本结构以及它的构建过程, 也正是因为这种倒排索引表结构的设计使得索引查询如此的高效。倒排索引也正是大多现代搜索引擎的基础。

Lucene 的内部倒排索引表构建过程大致如下:

假定我们有这样的三篇文档:

```
D0 = "What makes life dreary is the want of motive."
D1 = " I figure life is a gift and I don't intend on wasting it."
D2 = " Life is made up of small pleasures."
```

想要根据关键词能够搜索到包含该关键词的文档, 那么首先我们需要获取每篇文档都包含了哪些关键词, 一般会经过如下过程:

首先需要读取每篇文档的内容, 然后找出内容的所有单词, 也就是常说的分词处理。对于英文文档, 一般就是按照空格进行分词, 对于中文而言, 则需要使用相应的中文分词器进行特殊处理。得到所有单词后, 我们还需要剔除一些毫无意义的单词, 比如英文里的 “the” “are” “at” “in” “to” “is”, 这些词被称为停用词, 像中文里的停用词有 “的”, “是”, “我”。然后还需要将所有的单词统一转换成小写。由于英文单词还有时态之分, 所以我们还需要将还原到单词的原型, 比如 loved 还原成 love。最后我们还需要剔除掉其中的所有标点符号, 因为也不会有用户会将标点符号作为搜索关键词进行搜索。上面的一系列处理过程全部是由分词器完成, 处理完成后, 我们将得到如下信息:

```
文档 0: make, life, dreary, want, motive
文档 1: figure, life, gift, intend, waste
文档 2: life, make, small, pleasure
```

根据上述信息, 我们还只能知道每个文档包含了哪些单词, 即文档→词的映射, 而倒排索引表是词→文档的映射, 即反向信息。由于某个单词可能出现在多个文档中, 当用户根据某个单词进行搜索时, 可能会返回多个文档给用户, 这就涉及到多个文档的排列顺序, 我们自然是希望把跟用户输入的搜索关键词相关的文档优先排在前面展示给用户, 所以还需要统计每个单词在每个文档中的出现频率即 Term Frequency, 有时候也需要在返回的结果中将用户的搜索关键词进行高亮显示, 所以我们也许需要记录每个单词在文档中的出现位置即 Term Position 等。而 term 出现在哪个文档中这些信息是保存在 Term Dictionary 里, term 的出现频率信息是保存在词频文件里, 而 term 的位置信息是保存在位置文件里。统计完这些信息后, 我们将得到如下这样一个倒排索引结构:

```
make → 0,1,[5,9]      2,1,[8,11]
life → 0,1,[11,14]     1,1,[9,12]     2,1,[0,3]
gift → 1,1,[19,22]
... // 为了节省篇幅, 其他单词省略
```

0,1,[5,9] 依次表示文档的 id、词的出现次数即词频、词的出现位置即 Term Position，上面的整个信息又称 postinglist。此外 postinglist 中除了会包含文档 id，词频，词位置信息，还可以包含 payload 信息，Payload（元数据）诞生于 Lucene 的 2.2 版本，它是在 Lucene 2.1 索引文件格式的基础上扩展而来，提供了一种可以灵活配置的高级索引技术。Payload 允许用户在倒排索引中存储额外的自定义信息，根据此信息可以用于影响文档的最终评分。举个例子：

比如你有两个文档：

文档 1: I want buy a book.

文档 2: I want book a hotel.

假如你刚好是在做一个跟书籍有关的全文检索功能，而用户输入的搜索关键词是 book，此时用户的本意应该是想要看到跟书籍相关的信息，其他跟书籍无关的信息应该尽量排在后面展示，显然文档 2 的 book 是预订的意思，跟书籍无关，虽然两个文档都包含了单词 book，显然此时文档 1 应该优先排在文档 2 前面。这时候，payload 就派上用场了。你可以在两个 book 上存储额外的信息来干预两个文档的最终评分。比如这样：

文档 1: I want buy a book|0.9.

文档 2: I want book|0.1 a hotel.

这样，在你的倒排索引里，原本只有 3 列信息：文档 id，词频，位置信息，现在就多了一项 payload，变成 4 列信息，若你没有存储 payload 信息，第 4 列是不存在的。你的 postinglist 就变成这样：

book → 0,1,[13,16],[0.9]

到此，Lucene 的倒排索引表就基本构建完成了，这也是 Lucene 索引结构的最核心的部分。

为了减小索引文件的大小，Lucene 对索引还使用了压缩技术。首先，对词典文件中的关键词进行了压缩，关键词压缩为 <前缀长度，后缀>，例如：当前词为“中国人民”，上一个词为“中国人”，那么“中国人民”被压缩为 <3，民>。其次大量用到的是对数字的压缩，数字只保存与上一个值的差值，这样设计可以减小数字的长度，进而减少保存该数字需要的字节数，最终的目的是节省空间。

倒排索引表构建完成后，Lucene 首先根据用户输入的单词对 Term dictionary 词典进行二元查找，找到该 Term，通过词频文件指针可以读取到该 Term 对应的文档 ID，从而找到该搜索关键词在哪些文档中出现过，最终将结果返回给用户。当然返回结果文档给用户之前，中间还涉及文档的打分阶段，详细内容读者可以自行阅读《Lucene in action》这本书去了解学习。

## 2.5.2 Lucene 中常见术语详解

在学习 Lucene 的过程中，我想大家势必会被一些 Lucene 的术语说法给整蒙了，比如

norms、docValues、payload、termOffset、indexed、stored、tokenized 等。所以弄清楚这些术语背后表达的含义我觉得很有必要，这有助于你更好的学习 Solr，毕竟 Lucene 是 Solr 的基石。

- ❑ Indexed：表示是否需要创建索引，即是否需要添加到倒排索引表中，一般用来修饰 Field 域的，如果你不对某个域创建索引，那么意味着你将不能根据该域的域值进行全文检索；
- ❑ Stored：表示是否需要存储某个域的域值，一般表示是否需要将域值写入到磁盘上进行持久化，持久化的目的是为了查询的时候能再次获取返回给用户做展示。当然存储则意味着会增大索引文件的体积。如果你的索引目录是基于内存的，那 Stored 设置 true 还是 false 都没什么意义；
- ❑ Tokenized：表示是否需要某个域的域值进行分词操作，如果你设置为不分词，那么会把域值全部内容当作一个 Term 存入倒排索引表；
- ❑ Norms：即 Normalization 的缩写，翻译过来就是标准化的意思。其实这里表示是 Lucene 评分机制里的标准化因子，使用标准化因子来影响文档的最终评分。这就牵扯到 Lucene 的评分机制了。

评分机制是 Lucene 的核心部分之一。Lucene 默认是按照评分机制对每个 Document 进行打分，然后在返回结果中按照得分进行降序排序。内部的打分机制是通过 Query、Weight、Scorer、Similarity 这几个协作完成的。想要根据自己的业务对默认的评分机制进行干预来影响最终的索引文档的评分，那你必须首先对 Lucene 的评分公式有所了解：

$$\text{score}(q, d) = \text{coord}(q, d) \cdot \text{queryNorm}(q) \cdot \sum_{t \in q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t, d))$$

coord(q,d)：这里 q 即 Query, d 即 Document，表示指定查询项在 Document 中出现的频率，频率越大说明该 Document 匹配度越大，评分就越高。

queryNorm(q)：用来计算每个查询的权重的，从它的参数只有一个 q 就知道，它只是用来衡量每个查询的权重的，使每个 Query 之间也可以比较，（注意：它的计算结果会影响最终 document 的得分值，但它不会影响每个文档的得分排序，因为每个 Document 都会应用这个 Query 权重值。）默认它的实现数学公式如下：

$$\text{queryNorm}(q) = \text{queryNorm}(\text{sumOfSquaredWeights}) = \frac{1}{\text{sumOfSquaredWeights}^{1/2}}$$

queryNorm 的实现代码在 DefaultSimilarity 类中：

```
/** Implemented as <code>1/sqrt(sumOfSquaredWeights)</code>. */
@Override
public float queryNorm(float sumOfSquaredWeights) {
    return (float)(1.0 / Math.sqrt(sumOfSquaredWeights));
}
```

tf(t,d)：用来统计指定 Term t 在 document d 中的出现频率，出现次数越多说明匹配度越

高,得分自然就越高,默认实现如下所示:

```
@Override
public float tf(float freq) {
    return (float)Math.sqrt(freq);
}
```

$\text{idf}(t)$ : 统计出现 Term  $t$  的 document 的频率  $\text{docFreq}$ ,  $\text{docFreq}$  越小,  $\text{idf}$  越大, 则得分越高 (一个 Term 若只在几个 document 中出现, 说明这几个 document 稀有, 物以稀为贵)。

$t.\text{getBoost}()$ : 就是给 Term 设置权重值, 比如使用 `QueryParser` 语法表达式时可以这样: `java^1.2`。

$\text{norm}(t,d)$ : 主要分两部分: 一部分是 Document 的权重, 不过在 Lucene5 中 Document 的权重已经被取消了, 一部分是 Field 的 `boost`, 一部分是 field 中分词器分出来的 Token 个数因素, 个数越多, 匹配度越低, 就好比你在 1000000 个字符中匹配到一个关键字和在 10 个字符中匹配到一个关键字, Lucene 认为后者权重更大应该排在前面。我们常说的 `norms` 其实就是评分公式里的  $\text{norm}(t,d)$  部分。

上面各个子函数计算出来的分值再相乘求积得到最终得分。那为什么需要标准化因子呢? 我们知道, 在搜索的过程中需要把跟用户输入的关键词具有相关性的文档返回给用户, 相关性大的文档得分就高, 自然就排在前面。而这里的相关性在 Lucene 里是基于空间向量模型 (Vector Space Model) 实现的。在计算相关性之前, 要计算 Term Weight, 即某 Term 相对于某 Document 的重要性。在计算 Term Weight (权重) 时, 需要考虑如下几个问题:

1) 不同的文档重要性不同。有的文档重要些, 有的文档相对不重要, 比如对于码农来说, 在搜索书籍的时候, 我们可能更希望让计算机方面的书籍尽量排在前面, 而文学方面的书籍则靠后显示;

2) 不同的域重要性也可能不同。有的域重要一些, 如关键字、标题, 有的域不重要一些, 如摘要、描述等。同样一个词 (Term), 出现在关键字中应该比出现在摘要中更重要;

3) 根据词 (Term) 在文档中出现的频率来决定该词对文档的重要性, 有些不太合理。比如长的文档词在文档中出现的次数相对较多, 这样短的文档比较吃亏。比如一个词在一本几百页的书出现了 10 次, 在另外一篇不足 100 字的文章中出现了 9 次, 那前者就应该排在前面吗? 未必, 显然该词在不足 100 字的文章中能出现 9 次, 可见其对此文章的重要性。鉴于以上原因, Lucene 在计算 Term Weight 时, 都会乘上一个标准化因子 (Normalization Factor), 来减少上面三个问题的影响。

标准化因子 (Normalization Factor) 是会影响随后评分 (score) 计算的, Lucene 的评分计算一部分发生在索引过程中, 一般是与查询语句无关的参数如标准化因子。标准化因子 (Normalization Factor) 的计算公式如下:

$$\text{norm}(t, d) = \text{doc.getBoost()} \cdot \text{lengthNorm} \prod_{\text{field } f \text{ in } d \text{ named as } t} f.\text{getBoost}()$$

它包括三个参数:



1) Document boost: 此值越大, 说明此文档权重越大, 即表示此文档越重要;

2) Field boost: 此值越大, 说明此域的权重值越大即表示此域越重要;

3)  $\text{lengthNorm}(\text{field}) = (1.0 / \text{Math.sqrt}(\text{numTerms}))$ : 一个域中包含的 Term 总数越大, 则表明文档越长, 此值就越小, 反之文档越短, 此值越大。简而言之, lengthNorm 就是用于干预 Term 在文档中的总数对于 Term 在文档中的重要性的因子。

从上面的公式中, 我们知道, 影响 norm 标准化因子的因素有 3 个: 文档的权重、域的权重、文档中 Term 的总数。而 Solr 里的 omitNorms 即表示取消指定域 Field 的标准化因子, 也就意味着取消域的权重、文档的权重以及 Term 在文档中的出现总数对于文档最后评分的影响。同时, 取消 norms 还可以节省一部分内存。因为不用在搜索阶段为索引中的每篇文档的每个域都占用一个字节来保存 norms 信息了。但是对 norms 信息的禁用是必须全部域都禁用的, 一旦有一个域不禁用, 则其他禁用的域也会存放默认的 norms 值。由于为了加快 norms 的搜索速度, Lucene 是根据文档编号乘以每篇文档的 norms 信息所占用的大小来计算偏移量的, 中间少一篇文档, 偏移量将无法计算。也即 norms 信息要么都保存, 要么都不保存。在 Solr 内部一些不需要分词的基础数据类型的域都默认取消了 Norms。

❑ multiValued: 用于表示指定 Field 是否是一个多值域。我们知道, 域值一般是单个值, 但有时候可能我们的域值是一个 List 集合或者一个数组, 这时候该如何处理域值建立索引呢? 为了解决这个问题, 所以设计了多值域;

❑ termVectors: 表示是否对指定 Field 启用空间向量模型, 当你需要使用 FastVector-Highlighter 高亮器或者 MoreLikeThis 功能时, 你就需要启用 termVectors;

❑ termPositions: 表示是否记录 Field 的域值中的每个 Term 的位置信息即记录当前是文档中的第几个 term, 前提是你必须要先启用 termVectors;

❑ termOffsets: 表示记录 Field 的域值中的每个 Term 的位置偏移量, 所谓偏移量, 其实就是 Term 在文档中的起始位置和结束位置, 都是从零开始计算。比如 I like Java 这个字符串, 其中单词 Java 的位置偏移量就是 [7,10];

❑ PositionIncrement: 表示当前 Term 的位置与前一个 term 的位置之间的差值即位置增量, 比如: I like the girl with long hair (我喜欢那个长发女孩), 这里 girl 与 like 之间的位置增量为 2, 因为 like 的 position=1, girl 的 position=3。但如果考虑停用词的情况, 那么 like 和 girl 之间的 the 是停用词, the 会被剔除掉, 此时 girl 与 like 之间的位置增量为 1。假如两个 term 之间的 positionIncrement=0, 那么说明两个 Term 处于文档中同一个位置, 利用这个特性可以实现同义词功能;

❑ PositionIncrementGap: 表示两个 Term 之间的间隙, 这是 Solr Schema.xml 里 field 的配置属性, 一般用于解决多值域短语查询 phrase query 的。比如你有个多值域, 它有 2 个域值:

author: John Doe

author: Bob Smith

对于多值域而言，默认 `PositionIncrementGap=0`，也就意味着默认多值域的域值是直接拼接在一起的，即 `john doe bob smith`，也就是说你如果输入 `doe bob` 也是可以搜索到该文档的。如果你不希望用户输入 `doe bob` 能搜索到，那么你可以设置 `PositionIncrementGap=N`，`N` 是任意一个大于零的整数即可，这样用户就不能跨多个域值搜索了，只能在多值域的单个域值内实现关键字搜索了；

□ `precisionStep`：一般用于数字域的范围查询，默认值是 4，调整这个属性值可以提升数字域的查询性能。我们知道，在 Lucene 里，数字其实也是以 `String` 形式存储，只不过内部会经过一定的编码转换以及字符压缩。经过编码后的字符串也必须是有顺序的，且大小比较要符合实际数字的比较结果。比如 `num1>num2`，那么编码后的字符串需要满足 `str1 > str2`。`positionStep` 是用来分解编码后的字符串的，意思是每间隔几位创建一个前缀索引，比如编码后的字符串是 `0100,0011,0001,1010`，会被分解成“`0100,0011,0001,1010`”“`0100,0011,0001`”“`0100,0011`”“`0100`”。`precisionStep` 值越小，分解出来的前缀索引就越多，自然会增大索引体积，但数字范围查询速度更快，反之 `precisionStep` 值越大，索引体积越小，但数字范围查询速度越慢。

数字范围查询原理如图 2-44 所示。

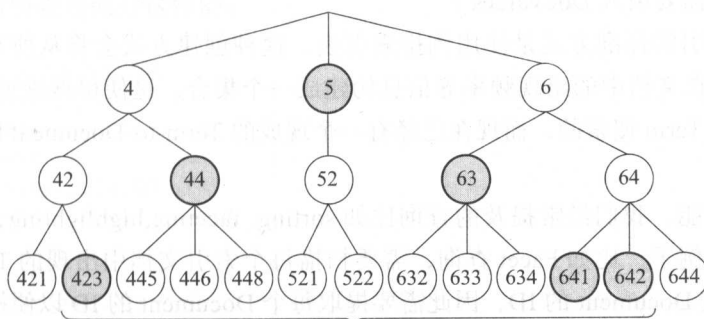


图 2-44 数字范围查询原理图

如果用户希望查找 423 ~ 642 的记录，如果没有 `precisionStep`，那么就必须从最底层一个一个匹配了，性能肯定不行。有了 `precisionStep`，那么首先对数字范围上限和下限按照 `precisionStep` 定义的位数进行分解，得到 4、42、423、5、6、64、642 这几个前缀索引，首先根据 423 和 642 这两个前缀索引可以直接找到底层的 423 和 642 这两个数字，然后根据前缀 4 找到前缀 44，再根据前缀 44 找到数字 445、446、448，同理根据前缀 5 可以找到数字 521、522，根据前缀 6 可以找到前缀 63，再根据前缀 63 可以找到数字 632、633、634，根据前缀 64 可以找到数字 641，至此数字范围查询就根据用户提供的数字范围找出了指定范围内的所有数字。

□ `DocValues`：Lucene 索引的存储一般都是以倒排索引的方式（`term-doc`），即 `Term` 到 `Document` 的一个映射。但是在搜索相关功能处理的时候，如排序、高亮，需要通过

文档 docid 找到相应的 term 值、term 的位置信息等。为此，在 Lucene4.0 中，引入了一个新字段类型 DocValue，即在索引的时候建立文档到值（document-to-value）的映射。这个方法保证减轻了一些字段缓存的内存要求，并且使得 Sorting、Faceting、Grouping、Function Query 的响应速度更快。但开启 DocValues 需要额外保存索引信息，因此会增大索引体积。

DocValues 只适用于部分 FieldType，这些 FieldType 底层实际又是使用的 Lucene 的 DocValues Type。DocValues 适用的 FieldType 如下所示：

#### 1) StrField 和 UUIDField:

- 如果这个 FieldType 是单值域（即 multi-valued=false），那么底层 Lucene 会使用 SORTED 类型；
- 如果这个 FieldType 是多值域（即 multi-valued=true），那么底层 Lucene 会使用 SORTED\_SET 类型。

#### 2) 所有以 Trie 开头的数字域，date 域，EnumField:

- 如果这个 FieldType 是单值域，那么底层 Lucene 会使用 NUMERIC 类型；
- 如果这个 FieldType 是多值域，那么底层 Lucene 会使用 SORTED\_SET 类型。

#### (1) 为什么需要引入 DocValues ?

Solr 创建索引的标准方式是使用倒排索引表。这种创建方式会将所有文档中找到的 Term 以及 Term 在文档中的出现频率等信息构建成一个集合，这使得搜索速度变得非常快。因为用户是通过 Term 搜索的，而现在已经有一个现成的 Term-to-Document 映射列表，所以查询处理很快。

对于其他功能，我们经常提及的查询比如 sorting, faceting, highlighting，这些查询并不是很高效，举个例子，比如 Facet 查询，需要扫描每个索引文档中出现的 Term，需要找到 Term 得实现知道 Document 的 ID，因此需要提取每个 Document 的 ID 以便构建 Facet 集合，但 Document ID 加载可能会导致磁盘 I/O，并且即便 Document ID 都加载到内存了，这些操作都在内存中进行，如果 Document 数目很大，那么数据加载会很慢。为此在 Lucene4.0 中，引入 DocValues 概念，DocValue 域是一个面向列的在索引时额外构建一个 document-to-value 的索引结构，这个方法可以减轻 fieldCache 对内存的需求并且使得 facet 查询、排序、Group 分组等这些操作变得更快。

#### (2) DocValues 的优缺点:

- 1) 近实时索引: 在每一个索引段里面都会有一个 docvalues 数据结构，这个结构与索引同时建立，并且能够快速更新、生效；
- 2) 基本的查询和过滤支持: 你可以做基本的词、范围等基本查询，但是不参与评分，并且速度较慢，如果你对速度和评分排序有要求，你可以讲该字段设置为 (indexed="true")；
- 3) 更好的压缩比: Docvalues fields 的压缩效果比 fieldcache 好，但不强调做到极致；
- 4) 节约内存: 你可以定义一个 FieldType 的 docValuesFormat (docValuesFormat="Disk")，

这样的只有一小部分数据加载到内存，其他部分保留在磁盘上。

**Payload**：所谓 payload 其实就是提供用户传入一个额外的自定义信息，而该信息可以干预文档最终的评分。Payload 信息跟 term 的位置 (positions) 信息一样，都是存储在倒排索引表中的，存储它会额外增大索引体积。我们知道，Lucene 中 Document 由 Field 组成，而 Field 由 Term 组成，文档的 Payload 可以用存储的 Field 表示。这样存在的问题是，如果需要读取大量的文档的元数据，因为 Field 的索引信息与存储信息是分开的，那么 I/O 效率将是较差的。而 Payload 信息则是直接存储在倒排索引中，检索出来是极快的，因此可以利用 Payload 功能存储文档级别的元数据，然后可以合理利用 payload 存储的数据来干预文档的评分，来完成一些特殊需求。

### (3) Payload 的应用场景举例

#### 场景一：改进的 Lucene 的区间检索

日期检索是区间检索的常见例子，比如用户需要在图书馆中检索特定年代的图书，满足如下条件：

date: [1984-08-01,1985-08-01]

常见的做法就是将日期作为一个独立 Field 进行存储，利用 RangeQuery 进行区间检索，此时你的倒排索引表可能是这样的：

```
Term          Postinglist
19230101—>1,2,10,22
19230102—>48
19550101—>56
19550102—>11,23,18,29
19550103—>67
.....
20080102—>66
20080103—>77,99
```

这个时候进行日期区间范围查询，我们只能从第一个日期扫描到最后一个日期，如果倒排索引表数目庞大，那么日期区间范围查询的性能自然就不好，这种情况下，我们可以利用 Payload 功能来减少词条的数目，提高检索效率，可以将日期的年月作为词条，作为 Payload 信息，这样可以减小索引体积，查询性能自然提升了。此时倒排索引表可以是这样的：

```
Term          Postinglist
192301—>1[1],2[1],10[1],22[1],48[2]
195501—>56[1],11[2],23[2],18[2],29[2],67[3]
.....// 省略
200801—>66[2],77[3],99[3]
```

#### 场景二：提高特定词汇的评分

比如你有这样两个文档：

文档 1: 北京小米科技有限责任公司成立 2010 年 4 月

文档 2: 位于北京市海淀区毛纺路 58 号院 3 号楼的小米科技

假定你有这样的需求: 当用户搜索“小米”时, 你期望文档 1 应该排在文档 2 前面, 因为文档 1 里的“小米”更靠前, 你觉得这篇文档与用户搜索关键词的相关性更强, 这个时候可以使用 payload 来实现。

首先你需要自定义一个支持解析 Payload 信息的分词器, 代码如下所示:

```
public class PayloadAnalyzer extends Analyzer {
    private PayloadEncoder encoder;
    PayloadAnalyzer(PayloadEncoder encoder) {
        this.encoder = encoder;
    }
    public TokenStream tokenStream(String fieldName, Reader reader) {
        TokenStream result = new WhitespaceTokenizer(reader);
        result = new LowerCaseFilter(result);
        result = new DelimitedPayloadTokenFilter(result, '|', encoder);
        return result;
    }
}
```

然后你需要索引的数据格式应该类似这样设置 Payload, 如下所示:

```
public static String[] DOCS = {
    "The quick|2.0 red|2.0 fox|10.0 jumped|5.0 over the lazy|2.0 brown|2.0 dogs|10.0",
    "The quick red fox jumped over the lazy brown dogs", //no boosts
    "The quick|2.0 red|2.0 fox|10.0 jumped|5.0 over the old|2.0 box|10.0",
    "Mary|10.0 had a little|2.0 lamb|10.0 whose fleece|10.0 was|5.0 white|2.0 as snow|10.0",
    "Mary had a little lamb whose fleece was white as snow",
    "Mary|10.0 takes on Wolf|10.0 Restoration|10.0 project|10.0 despite ties|10.0 to sheep|
10.0 farming|10.0",
    "Mary|10.0 who lives|5.0 on a farm|10.0 is|5.0 happy|2.0 that she|10.0 takes|
5.0 a walk|10.0 every day|10.0",
    "Moby|10.0 Dick|10.0 is|5.0 a story|10.0 of a whale|10.0 and a man|10.0 obsessed|10.0",
    "The robber|10.0 wore|5.0 a black|2.0 fleece|10.0 jacket|10.0 and a baseball|10.0
cap|10.0",
    "The English|10.0 Springer|10.0 Spaniel|10.0 is|5.0 the best|2.0 of all dogs|10.0"
};
```

Payload 信息设置格式为 term|payload 信息, 中间必须使用这种中竖线|进行分割, 因此一开始我们示例文档数据应该这样设置 Payload:

文档 1: 北京小米 |0.9 科技有限责任公司成立 2010 年 4 月

文档 2: 位于北京市海淀区毛纺路 58 号院 3 号楼的小米 |0.1 科技

然后你需要继承 DefaultSimilarity 实现自己的 Similarity 打分器, 示例代码如下所示:

```
public class PayloadSimilarity extends DefaultSimilarity {
    @Override
    public float scorePayload(String fieldName, byte[] bytes, int offset, int length) {
```

```

        return PayloadHelper.decodeFloat(bytes, offset);
    }
}

```

然后执行索引查询，示例代码如下：

```

IndexSearcher searcher = new IndexSearcher(dir, true);
searcher.setSimilarity(payloadSimilarity);
BoostingTermQuery btq = new BoostingTermQuery(new Term("body", "fox"));
TopDocs topDocs = searcher.search(btq, 10);
for (int i = 0; i < topDocs.scoreDocs.length; i++) {
    ScoreDoc doc = topDocs.scoreDocs[i];
    System.out.println("Doc: " + doc.toString());
    System.out.println("Explain: " + searcher.explain(btq, doc.doc));
}

```

上述演示的是如何在 Lucene 中使用 Payload 来实现提高特定词汇评分的功能，其实在 Solr 中实现更简单。

首先你需要在 schema.xml 中配置能支持 Payload 信息解析的 fieldType，配置示例如下所示：

```

<fieldtype name="payloadField" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.DelimitedPayloadTokenFilterFactory"
encoder="integer" delimiter="|"/>
</analyzer>
// 设置自定义的打分器工厂类 PayloadSimilarityFactory，具体实现看下面
<similarity class="payloadexample.PayloadSimilarityFactory" />
</fieldtype>

```

在指定 field 域上应用我们刚刚定义的 payloadField，配置示例如下所示：

```

<field name="speech" type="payloadField" indexed="true" stored="true"
multivalued="true" />

```

你需要自定义自己的打分器工厂类，示例代码如下所示：

```

package payloadexample;

import org.apache.lucene.analysis.payloads.PayloadHelper;
import org.apache.lucene.search.similarities.DefaultSimilarity;
import org.apache.lucene.search.similarities.Similarity;
import org.apache.lucene.util.BytesRef;
import org.apache.solr.common.params.SolrParams;
import org.apache.solr.schema.SimilarityFactory;

public class PayloadSimilarityFactory extends SimilarityFactory {
    @Override
    public void init(SolrParams params) {
        super.init(params);
    }
}

```



```

@Override
public Similarity getSimilarity() {
    return new PayloadSimilarity();
}

}

class PayloadSimilarity extends DefaultSimilarity {
    @Override
    public float scorePayload(int doc, int start, int end, BytesRef payload) {
        if (payload == null) return 1.0F;
        return PayloadHelper.decodeFloat(payload.bytes, payload.offset);
    }
}

```

在你的 schema.xml 末尾处配置上自定义的 Similarity，配置示例如下所示：

```

<similarity class="org.apache.lucene.search.similarities.DefaultSimilarity"/>
</schema>

```

然后你需要自定义 QueryParser 和 QParserPlugin 插件：

```

package payloadexample;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.payloads.AveragePayloadFunction;
import org.apache.lucene.search.payloads.PayloadTermQuery;
import org.apache.solr.common.params.CommonParams;
import org.apache.solr.common.params.SolrParams;
import org.apache.solr.common.util.NamedList;
import org.apache.solr.parser.QueryParser;
import org.apache.solr.request.SolrQueryRequest;
import org.apache.solr.schema.SchemaField;
import org.apache.solr.search.QParser;
import org.apache.solr.search.QParserPlugin;
import org.apache.solr.search.QueryParsing;
import org.apache.solr.search.SyntaxError;
public class PayloadQParserPlugin extends QParserPlugin {
    @Override
    public void init(NamedList args) {}
    @Override
    public QParser createParser(String qstr, SolrParams localParams, SolrParams
params, SolrQueryRequest req) {
        return new PayloadQParser(qstr, localParams, params, req);
    }
}

class PayloadQParser extends QParser {
    PayloadQueryParser pqParser;
    public PayloadQParser(String qstr, SolrParams localParams, SolrParams params,
SolrQueryRequest req) {
        super(qstr, localParams, params, req);
    }
}

```

```

@Override
public Query parse() throws SyntaxError {
    String qstr = getString();
    if (qstr == null || qstr.length() == 0) return null;
    String defaultField = getParam(CommonParams.DF);
    if (defaultField == null) {
        defaultField = getReq().getSchema().getDefaultSearchFieldName();
    }
    pqParser = new PayloadQueryParser(this, defaultField);
    pqParser.setDefaultOperator
        (QueryParsing.getQueryParserDefaultOperator(getReq().getSchema(),
            getParam(QueryParsing.OP)));
    return pqParser.parse(qstr);
}

@Override
public String[] getDefaultHighlightFields() {
    return pqParser == null ? new String[]{} :
new String[] {pqParser.getDefaultField()};
}

// 注意这里不支持 phrase queries
class PayloadQueryParser extends QueryParser {
    PayloadQueryParser(QParser parser, String defaultField) {
        super(parser.getReq().getCore().getSolrConfig().luceneMatchVersion, default-
Field, parser);
    }

    @Override
    protected Query getFieldQuery(String field, String queryText, boolean quoted)
throws SyntaxError {
        SchemaField sf = this.schema.getFieldOrNull(field);
        if (sf != null && sf.getType().getTypeName().equalsIgnoreCase("payload-
Field")) {
            // 这里 AveragePayloadFunction 表示当你的文档中有多个相同的 term, 而每个 term 都
            有 payload 信息, 而 payload 里存储的是每个 term 的权重值, 所以这里是求这些名称相同的 term 的 payload
            平均值作为该 term 在文档中的权重值
            return new PayloadTermQuery(new Term(field, queryText), new AveragePayload-
Function(), true);
        }
        return super.getFieldQuery(field, queryText, quoted);
    }
}

```

然后你需要在 solrconfig.xml 中注册刚刚自定义的 QParserPlugin 插件, 配置示例如下所示:

```

<dataDir>C:\solr_home\core1\data</dataDir>
<lib dir="./lib" regex=".*\.jar"/>
<queryParser name="myqp" class="payloadexample.PayloadQParserPlugin" />

```

到此, Solr 中的 Payload 使用就介绍完了。

### 2.5.3 创建 Solr 索引

经过上一节的学习，我们了解了 Lucene 的索引创建原理，下面我们需要学习在 Solr 中如何创建和更新索引，其实，更准确地说，Solr 根本不存在索引更新操作，因为 Lucene 更新索引本质就是先删除再创建。前面的章节我们已经了解了如何使用 Solr DIH 来导入数据创建 Solr 索引。由于 Solr 内部提供了 /update 接口来实现索引数据的提交和创建，这个功能已经集成到 Solr 的 Web 后台中，如图 2-45 所示。

update 接口 URL 为：<http://localhost:8080/solr/core/update>。

这个接口可以实现索引文档的添加删除更新，比如根据文档 ID 删除索引数据：

```
http://localhost:8080/solr/core/update/?stream.body=<delete><id>id 值 </id></delete>&stream.contentType=text/xml;charset=utf-8&commit=true
```

这里的 stream.body 参数的内容必须符合 Solr 规定的格式，之前的章节有做说明，这里就不再重复了，如果是添加索引，那么就是使用 <add> 元素，需要注意的是 stream.body 的参数值需要进行 URL 编码，比如一些特殊字符 >< 需要编码成 %3E%3C，如果是更新索引，你需要为 <doc id=“xxx”> 添加文档 ID，这样只要该文档 ID 在索引中存在，则会执行索引更新操作，否则执行索引创建操作。stream.body 在 http 协议里其实就是 request payload。你完全可以使用 HttpClient 来模拟 Http Request payload 参数，示例代码如下所示：

```
StringEntity se = new StringEntity(payloadString);
httpPost.setEntity(se);
```

上面的 payloadString 即接口 URL 里 stream.body 参数值，比如这样：

```
{
  add:
  {
    doc: {
      id: "1", title: "change.me"
    },
    boost: 1,
    overwrite: true,
    commitWithin: 1000
  }
}
```

后续会详细讲解如何使用 SolrJ API 来创建删除更新 Solr 索引，这里暂时只做了解。



图 2-45 Solr 索引 Update 功能界面

## 2.5.4 Solr Cell

一般用户都会需要能够接收二进制的结构化文件（比如 word 文件，PDF 等等）并对其创建索引。Apache Tika 框架内部包装了很多不同格式的转换器，比如 PDFBox、POI。Solr 的 ExtractingRequestHandler 使用 Tika 允许用户上传二进制文件，Solr 然后从中提取文本建立索引，这就是 Solr Cell。

开始学习 Solr Cell 之前，我们需要了解几个概念来帮助我们更好的理解 Solr Cell。

- Tika 会自动检测输入文档的类型，然后采用适当的方式提取文件的文本内容，当然你也可以通过 stream 参数显式的指定文件的 MIME 类型；
- Tika 通过 SAX ContentHandler 生成一个 XHTML 流，然后 Solr 响应 Tika 的 SAX 事件，然后创建域并索引；
- Tika 还会生成额外的 Metadata 元数据信息，比如 Title, Subject, Author；
- Tika 提取出来的所有文本最终都是添加到名称为 content 的 Solr 域上；
- 我们可以把 Tika 提取的元数据信息也映射到 Solr 的域上；
- 我们可以通过 literals.fieldName 这种方式为文档添加额外的域，补充 Tika 通过提取 metadata 元数据无法获取到的那部分数据；
- 我们也可以使用 XPath 表达式来帮助 Tika 更好地去提取内容。

要想使用 Solr Cell 功能，首先你需要在 core/lib 目录下添加 Solr Cell 依赖的 jar 包：apache-solr-cell.jar 以及 contrib\extraction\lib 目录下的所有 jar 包。

然后需要在 solrconfig.xml 里配置 ExtractingRequestHandler 处理器，配置示例如下所示：

```
<requestHandler name="/update/extract" class="org.apache.solr.handler.extraction.
ExtractingRequestHandler">
  <lst name="defaults">
    <str name="fmap.Last-Modified">last_modified</str>
    <str name="uprefix">ignored_</str>
  </lst>
  <!-- 配置 tika 配置文件的加载路径 -->
  <str name="tika.config">/my/path/to/tika.config</str>
  <!-- 配置日期格式 -->
  <lst name="date.formats">
    <str>yyyy-MM-dd</str>
  </lst>
</requestHandler>
```

当你需要上传大文件时，那么还需要调整 multipartUploadLimitInKB 参数值，在 solrconfig.xml 中添加如下配置：

```
<requestDispatcher handleSelect="true" >
  <requestParsers enableRemoteStreaming="{true|false}" multipartUploadLimitInKB=
"2048000" />
```

Solr Cell 的 Http 接口 URL:

`http://localhost:8080/solr/update/extract?stream.file=/path/to/file/StatesLeftToVisit.doc&stream.contentType=application/msword&literal.id=states.doc`

`stream.file` 表示你的富文本文件路径, `stream.contentType` 表示你富文本文件的 MIME 类型, 如果富文本文件不在本地机器上, 那么还需要添加 `enableRemoteStreaming=true` 参数来开启对远程文件流的加载。

为了使 Tika 在提取文本内容时能够自动将日期格式的字符串转换成 `java.util.Date` 类型, Solr Cell 提供了 `date.formats` 参数供用户来指定日期格式, 内置支持的日期格式如下所示 (具体请查阅 Solr 的 `DateUtil` 类)

```
yyyy-MM-dd'T'HH:mm:ss'Z'
yyyy-MM-dd'T'HH:mm:ss
yyyy-MM-dd
yyyy-MM-dd hh:mm:ss
yyyy-MM-dd HH:mm:ss
EEE MMM d hh:mm:ss z yyyy
EEE, dd MMM yyyy HH:mm:ss zzz
EEEE, dd-MMM-yy HH:mm:ss zzz
EEE MMM d HH:mm:ss yyyy
```

如果你使用了多个 Core, 那么多个 Core 之间可能会有重复的 jar 包, 此时你可以在 `solr.xml` 里配置 `sharedLib` 属性, `sharedLib` 用于指定所有 core 共享的 lib 目录, 配置示例如下所示:

```
<solr sharedLib="lib">
<solrcloud>
```

`sharedLib="lib"` 这里的 `lib` 表示所有 core 的 jar 包共享目录为当前 `solr.xml` 同级目录下的 `lib` 目录即它是一个相对路径, 默认是相对 `#{SOLR_HOME}`。当然你也可以将 `sharedLib` 属性值配置为一个绝对路径。

Tika 会隐式的为每个文档提取额外的元数据信息, 比如作者信息、当前页码等, 对于提交的不同文件类型, Tika 提取的 `metadata` 元数据信息不尽相同, 除了 Tika 自身会提取元数据信息之外, Solr 自身也会添加以下元数据信息 (全部定义在 `ExtractingMetadataConstants` 常量类里):

- `"stream_name"`: 上传的文件源文件名, 不一定有值, 这取决于文件是以何种方式上传的;
- `"stream_source_info"`: 关于上传的文件流的源信息, 具体请看 Solr 的 `ContentStream` 类;
- `"stream_size"`: 文件流的字节大小;
- `"stream_content_type"`: 文件流的 `content-type` 类型。

有时候可能遇到某个 PDF 文件加密了, 此时你可以在请求参数里添加 `resource.password=` 你的文件密码来解决, 当然也可以提供一个密码配置文件, 将加密文件的密码配置在文件

里, 这种方式适合于加密文件较多的情况, 你可以使用正则表达式来批量设置密码, 密码配置文件配置示例如下所示:

```
myFileName = myPassword
.*\docx$ = myWordPassword
.*\pdf$ = myPdfPassword
```

第一种方式的接口请求 URL 示例如下:

```
http://localhost:8080/solr/core/update/extract?commit=true&literal.id=123&resource.
password=mypassword
```

第二种通过密码配置文件方式的接口请求 URL 示例如下:

```
http://localhost:8080/solr/core/update/extract?commit=true&literal.id=123&passwords
File=myspass.properties&resource.name=my-encrypted-file.pdf
```

passwordFile 参数用于指定密码配置文件的本地硬盘加载路径。

可以通过以下 3 种方式发送 ContentStream 给 ExtractingRequestHandler 进行处理:

- 1) Raw POST: 表示数据是以 text、json、xml、html 这几种格式传递的;
- 2) Multi-part file upload: 上传的文件(每个文件会被处理生成一个唯一的 document)

通过 "stream.body"、"stream.url"、"stream.file" 请求参数传递;

- 3) 使用 SolrJ 上传文件并对文件创建索引, 示例代码如下所示:

```
import java.io.File;
import java.io.IOException;
import org.apache.solr.client.solrj.SolrServer;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.request.AbstractUpdateRequest;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.impl.CommonsHttpSolrServer;
import org.apache.solr.client.solrj.request.ContentStreamUpdateRequest;
public class SolrExampleTests {
    public static void main(String[] args) {
        try {
            String fileName = "c:/Sample.pdf";
            String solrId = "Sample.pdf";
            indexFilesSolrCell(fileName, solrId);
        } catch (Exception ex) {
            System.out.println(ex.toString());
        }
    }
}
```

/\*\*

\* Method to index all types of files into Solr.

\* @param fileName

\* @param solrId

\* @throws IOException



```
* @throws SolrServerException
*/
public static void indexFilesSolrCell(String fileName, String solrId)
throws IOException, SolrServerException {
    String urlString = "http://localhost:8080/solr";
    SolrServer solr = new CommonsHttpSolrServer(urlString);
    ContentStreamUpdateRequest up
        = new ContentStreamUpdateRequest("/update/extract");
    up.addFile(new File(fileName));
    up.setParam("literal.id", solrId);
    up.setParam("uprefix", "attr_");
    up.setParam("fmap.content", "attr_content");
    up.setAction(AbstractUpdateRequest.ACTION.COMMIT, true, true);
    solr.request(up);
    QueryResponse rsp = solr.query(new SolrQuery("*:~"));
    System.out.println(rsp);
}
}
```

如果需要设置多值域，可以使用 `ModifiableSolrParams` 类实现，代码如下所示：

```
ModifiableSolrParams p = new ModifiableSolrParams();
for(String value : values) {
    p.add(ExtractingParams.LITERALS_PREFIX + "field", value);
}
up.setParams(p);
```

## 2.5.5 Solr 索引去重检测

Solr 索引去重检测（即 Solr Document Duplication Detection），其实就是在我们创建索引文档时，就根据指定的一个或多个域的域值按照一定的算法生成一个签名摘要或者指纹，然后将这个签名摘要或者指纹当作一个独立的域添加到当前 Document 中，这个域被称为 `SignatureField`（签名域），它就可以作为 Document 的唯一性标识，用户可以将这个签名域在 `schema.xml` 中配置为 `uniqueKey`（唯一主键），由于 Solr 在创建索引时，如果发现 document 的 `uniqueKey` 相同就会执行索引更新操作。其实如果你的业务数据已经包含了明显的主键，那么通过生成签名域来确保 Document 的唯一性就显得不那么重要了。生成签名需要时间，而额外添加一个签名域无疑增大了你的索引体积，所以这种 Solr 索引去重检测可以在你的业务数据没有主键的时候使用。

Solr 为生成唯一签名内置提供了 3 种实现如表 2-21 所示。

表 2-21 Solr 生成签名实现方式

类名	描 述
MD5Signature	基于 MD5 实现的 128 位 hash 散列
Lookup3Signature	基于 Lookup3 实现的 64 位 hash 散列，它比 MD5 生成速度稍快
TextProfileSignature	来自于 Nutch 内部的模糊 Hash 散列实现，它更适合于对长文本生成签名

想要配置启用 Solr Document Duplication Detection 功能, 首先你需要在你的 solrconfig.xml 中配置 updateRequestProcessorChain 即更新请求处理器链条, 在这个处理器链条里配置好索引更新请求需要经过哪几个 updateRequestProcessor 更新处理器, updateRequestProcessor 严格按照配置的顺序从上至下依次执行。配置示例如下所示:

```
<updateRequestProcessorChain name="dedupe">
  <processor class="org.apache.solr.update.processor.SignatureUpdateProcessorFactory">
    <bool name="enabled">true</bool>
    <bool name="overwriteDups">false</bool>
    <str name="signatureField">id</str>
    <str name="fields">name,features,cat</str>
    <str name="signatureClass">org.apache.solr.update.processor.Lookup3Signature</str>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

上述配置中, 在更新请求处理器链条里配置了 3 个更新请求处理器: SignatureUpdateProcessorFactory、LogUpdateProcessorFactory、RunUpdateProcessorFactory。

LogUpdateProcessorFactory 用于记录程序日志, RunUpdateProcessorFactory 是真正用于处理 Solr 索引更新请求的处理器 (这里的 update 其实包含了 add/update/delete 这 3 个类型操作)。

SignatureUpdateProcessorFactory 提供了几个可选的配置参数如表 2-22 所示。

表 2-22 SignatureUpdateProcessorFactory 提供的配置参数

属性	描 述
signatureClass	指定生成签名的实现类, 必须是类额完整包路径, 若未指定, 默认值为 org.apache.solr.update.processor.Lookup3Signature
fields	根据哪几个域的域值来生成签名, 多个域使用逗号分隔, 不指定默认就是文档的所有域
signatureField	生成的签名域的名称, 该域需要在 schema.xml 中提前定义, 若为配置, 默认值为 signatureField
enabled	是否启用索引去重检测, 默认为 true 即启用

然后你需要在 schema.xml 中定义签名域, 域名称随意取, 不重复即可。但要跟 <str name="signatureField">id</str> 这里配置的域名保持一致。配置示例如下所示:

```
<field name="id" type="string" stored="true" indexed="true"
multiValued="false" />
```

最后你需要在 solrconfig.xml 中配置 requestHandler: 配置示例如下所示:

```
<requestHandler name="/update" class="solr.UpdateRequestHandler" >
  <lst name="defaults">
    <str name="update.chain">dedupe</str>
  </lst>
</requestHandler>
```

配置 requestHandler 的同时指定它的请求处理链条为 dedupe。

### 2.5.6 Solr 更新请求处理链

我们的索引创建,更新,删除操作内部都是交由 UpdateRequestProcessor 处理的,虽然最终提供给用户使用的是 UpdateRequestHandler,但 UpdateRequestHandler 其实就是依赖 UpdateRequestProcessor 来完成处理的,而 UpdateRequestProcessor 都是注册在 UpdateRequestProcessorChain 链条上的,UpdateRequestHandler 通过 UpdateRequestProcessorChain 链条获取需要经过哪些 UpdateRequestProcessor 处理。而我们并没有在 solrconfig.xml 里配置任何 UpdateRequestProcessorChain 或 UpdateRequestProcessor 啊,那它们是如何初始化的呢?其实在 Solr Core 加载的时候就会去初始化 Solr 的插件,在初始化插件函数里会检查用户是否在 solrconfig.xml 里配置 <updateRequestProcessorChain> 元素来定义 UpdateRequestProcessor 执行链条,如果用户没有配置,那么 Solr 会创建一个默认的 UpdateRequestProcessorChain,具体看 SolrCore 类的源码可以得知,如下代码所示:

```
if (def == null) {
    log.info("no updateRequestProcessorChain defined as default, creating
implicit default");
    //construct the default chain
    UpdateRequestProcessorFactory[] factories = new UpdateRequestProcessor-
Factory[]{
        new LogUpdateProcessorFactory(),
        new DistributedUpdateProcessorFactory(),
        new RunUpdateProcessorFactory()
    };
    def = new UpdateRequestProcessorChain(Arrays.asList(factories), this);
}
```

默认会创建一个 UpdateRequestProcessorChain 并注册 3 个 UpdateProcessorFactory,其中 DistributedUpdateProcessorFactory 用于 Solr Cloud 分布式模式下的索引更新请求处理。当然我们也可以显示的在 solrconfig.xml 里配置 UpdateRequestProcessorChain,具体配置如下代码所示:

```
<updateRequestProcessorChain name="mychain" default="true">
<processor class="com.example.MyCustomProcessorFactory" >
<lst name="name">
<str name="n1">x1</str>
<str name="n2">x2</str>
</lst>
</processor>
<processor class="solr.LogUpdateProcessorFactory" />
<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

其中 MyCustomProcessorFactory 是你的自定义 UpdateProcessorFactory 配置,配置的更

新处理器工厂会严格按照配置的顺序从上至下依次执行。

然后我们可以配置 `<update>` 并指定 `UpdateRequestProcessorChain`，配置示例如下所示：

```
<requestHandler name="/update/processortest" class="solr.JsonUpdateRequestHandler" >
<lst name="defaults">
<str name="update.chain">mychain</str>
</lst>
</requestHandler>
```

2.5.7 Solr 原子更新

当你将需要索引的数据索引到 Solr 中之后，你可能就会开始思考：我该采用什么策略来更新 Solr 中的索引文档？Solr 提供了两个方式来完成索引文档的部分更新。

第一种方式就是 Atomic Update（原子更新）。这种方式允许你只更新索引文档中的一个或多个域，而不需要对整个索引进行重建。原子更新使得每次只更新索引文档的部分数据成为可能。

第二种方式就是 optimistic concurrency（并发乐观锁），它是很多 NoSQL 数据库的一个功能特性，它允许根据版本号有条件的更新索引文档，这种方式包含了如何处理版本匹配的语义和规则。原子更新和并发乐观锁既可以独立的管理文档的更新，也可以结合在一起使用。但需要注意的是，并发乐观锁只是保证了对于同一个索引文档不会出现两个索引更新操作同时并发执行，但对索引更新操作本身是不是原子更新并不关心。

1. 原子更新

Solr 提供了多种 modifier（修改器）来自动更新文档的值，它允许只更新文档的指定域，这会帮助你提高索引创建速度，对于文档更新频率呈指数增长的应用程序来说，原子更新无疑是巨大的性能优化。

为一个需要更新的域添加一个 modifier（修改器），这样它就可以使用原子更新，域值可以被更新、添加、删除，如果域值是数字，它还可以进行递增。原子更新支持如表 2-23 所示操作。

表 2-23 原子更新支持操作

Modifier	描 述
set	设置域值，你也可以设置为 NULL 来删除域值。
add	为多值域追加值，可以是单个值也可以是一个集合。
remove	移除多值域的值，可以指定单个值也可以指定一个集合。
removeregex	同 remove 类似，removeregex 根据指定的正则表达式移除匹配到的值。
inc	按指定的数值递增域值，用于数字域。

要想使用原子更新，那么要求你的域必须配置 `stored="true"`，`copyField` 除外。原子更新只能应用于 `stored="true"` 的存储域上。如果 `CopyField` 的 `dest` 域也配置了 `stored="true"`，`CopyField` 的域值有一部分可以来自于 `source` 域，而 `source` 域的数据可能来自索引创建程序

输入的, CopyField 的域值也有一部分可能来自其他 CopyField, 那么当对 CopyField 执行原子更新, 来自索引创建程序的那部分索引数据可能会丢失。假如你有这样一个 Document:

```
{
  "id": "mydoc",
  "price": 10,
  "popularity": 42,
  "categories": ["kids"],
  "promo_ids": ["a123x"],
  "tags": ["free_to_try", "buy_now", "clearance", "on_sale"]
}
```

执行下面的原子更新操作:

```
{
  "id": "mydoc",
  "price": {"set": 99},
  "popularity": {"inc": 20},
  "categories": {"add": ["toys", "games"]},
  "promo_ids": {"remove": "a123x"},
  "tags": {"remove": ["free_to_try", "on_sale"]}
}
```

原子更新操作执行之后, Document 变成这样:

```
{
  "id": "mydoc",
  "price": 99,
  "popularity": 62,
  "categories": ["kids", "toys", "games"],
  "tags": ["buy_now", "clearance"]
}
```

上面的示例都是使用 JSON 格式完成原子更新, 同理你也可以使用 XML 格式来完成原子更新, 具体示例如下所示:

```
<add>
  <doc>
    <field name="id">123</field>
    <field name="author" update="set">Tom</field>
  </doc>
</add>
```

## 2. 并发乐观锁

Solr 客户端在请求更新 Document 时可以使用并发乐观锁来保证该 Document 没有被其他客户端并发更新。这个功能需要在你所有 Document 上添加一个 `_version_` 域, 对照 `_version_` 域也是索引更新操作的一部分, 默认 Solr 在 `shcema.xml` 里已经预定义了一个 `_version_` 域, 并且这个域是自动添加到索引文档中的, 用户不需要做任何干预。如果你想要禁用 `_version_`

域,那么你必须同时禁用 Solr 的 update log 功能,即注释掉 solrconfig.xml 的 <updateLog> 配置,否则 Solr Server 无法启动。但是,如果你使用 SolrCloud,那么 \_version\_ 和 update log 必须都启用。如果一个 Document 没有 \_version\_ 域,那么原子更新功能将无法正常工作。

当 Client 客户端 POST 一个索引文档到 Solr Server,Client 可以添加一个可选参数 versions=true 来告诉 Solr Server,请在响应信息里带上被添加进去的索引文档的最新版本号。具体示例如下:

```
POST http://localhost:8080/solr/core/update?versions=true
[
  { "id" : "aaa" },
  { "id" : "bbb" }
]
```

// Response

```
{
  "responseHeader":{"status":0,"QTime":6},
  "adds":[
    "aaa",1498562471222312960,
    "bbb",1498562471225458688
  ]
}
```

默认 Solr 的 schema.xml 中已经预先定义了一个 \_version\_ 域,并且这个域会自动添加到每个索引文档中:

```
<field name="_version_" type="long" indexed="true" stored="true"/>
```

如果你的索引文档需要频繁的更新或者你的索引数据量很大以致于没有足够的内存提供给 FieldCache 来缓存 \_version\_ 域,此时比较好的解决方案是使用 docValues 代替 indexed。配置示例如下所示:

```
<field name="_version_" type="docval_long" indexed="false" stored="true" required="true" docValues="true"/>
<fieldType name="docval_long" class="solr.TrieLongField" precisionStep="0" positionIncrementGap="0" docValuesFormat="Disk"/>
```

上面的 docValuesFormat=“Disk”属性表示当 FieldCache 的内存不够用时,会将缓存数据溢写到硬盘上,从而缓解内存压力。docValuesFormat 属性自 Solr 4.9 版本开始已经移除,因为它的溢写操作带来的 I/O 操作性能低下。

## 2.5.8 使用 Luke 查看索引

Luke 是一个基于 Lucene 实现的一个方便开发和诊断的小工具,使用它你可以很方便地访问 Lucene 索引以及允许你显示修改查询索引数据。Luke 提供的功能如下:

- 根据文档编号或 term 来浏览索引文档;



- ☐ 可以查看或复制索引文档到剪切板；
- ☐ 检索列表中出现频率最高的 Term；
- ☐ 执行索引查询并浏览返回的查询结果；
- ☐ 分析搜索结果；
- ☐ 删除索引文档；
- ☐ 支持重构原有的索引域，重新编辑修改并重新插入到索引；
- ☐ 支持优化索引功能。

Luke 下载地址：<https://github.com/DmitryKey/luke/releases>，如果你想要下载 Luke 的源码进行学习研究，那么你可以下载 Source core 压缩包。这里我下载的是 luke-5.3.0 的 luke-with-deps.zip 压缩包。解压后直接双击 luke.bat 即可运行，luke.sh 脚本是用于在 Linux 环境下运行 Luke，Luke 运行成功后，会弹出一个界面要求用户选择一个索引目录，如图 2-46 所示。

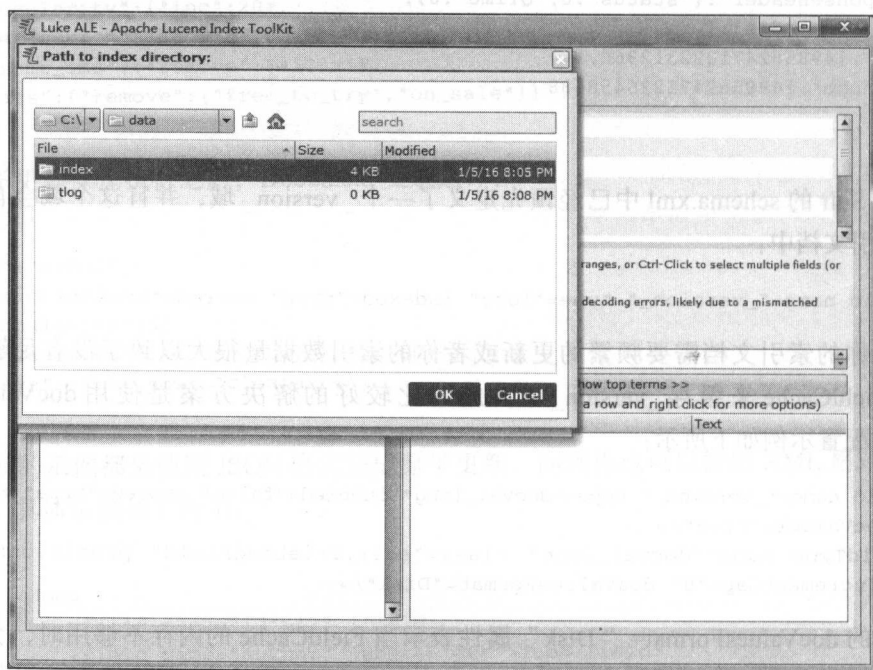


图 2-46 选择索引目录

运行后，你可以通过 Luke 查看索引文档、文档的搜索测试以及分词情况，在开发阶段，Luke 算是一个不错的开发测试工具，当然 Solr 的 Web Application 也是一个不错的选择。

Luke 的使用比较简单，虽然是纯英文的，但我相信你们应该能学会怎么使用各功能。它的主要功能如图 2-47 所示（依次是概述、文档管理、文档查询、分词测试、索引文件预览、插件）。

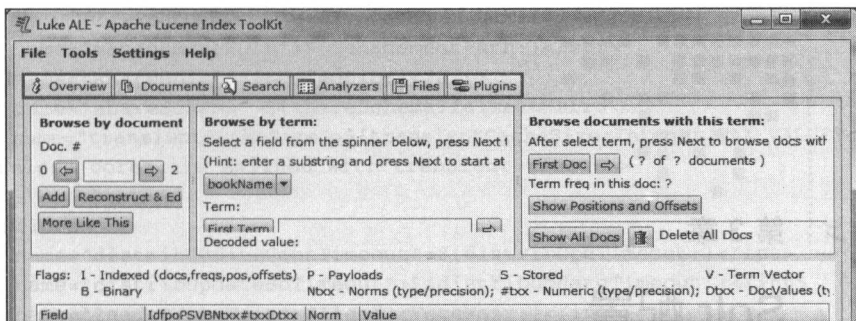


图 2-47 Luke 功能

## 2.6 本章总结

在本章中，我们主要认识一下什么是 Solr Core，以及了解 Core 的基本原理操作，粗略熟悉一下 Solr 提供的用于管理 Core 的 Http 接口，并学会如何通过 Solr 的后台管理 UI 来操作 Core。然后我们学会了如何熟练地使用 Solr DIH 以及 Solr 的全量和增量导入功能。紧随其后的就是本章的重点，即熟悉 Solr 索引的基本原理，之所以提前在第 2 章做讲解，是因为这是学习 Solr 的核心基础。考虑到有些同学是直接跳过 Lucene，简单粗暴的直面 Solr，因此我希望这部分同学能着重加强理解本章提及或涉及的一些概念术语，如果在学习过程中在概念理解上仍有一些疑惑，你仍然可以随时翻回到本章进行查阅。

## Solr 配置

为了 Solr 框架自身使用的灵活性和可扩展性，Solr 提供了各种配置文件来支撑并满足各项功能需求，并使可配置化、可定制化成为可能，用户甚至不需要了解 Solr 内部运行机制和 Lucene 基础，即可轻松使用 Solr。但为了更好更高效地使用 Solr，熟悉 Solr 的各项配置文件就成了必修之课，这样你完成一些企业项目也能更得心应手。

通过第 3 章，你将可以学习到如下内容：

- 熟悉 solr.xml 配置；
- 熟悉 solrconfig.xml 配置；
- 熟悉 schema.xml 配置；
- 熟悉 data-config.xml 配置。

### 3.1 solr.xml 配置详解

solr.xml 主要用于配置 Core 相关的管理参数设置，比如 Core 的加载线程数、Core 的根目录、Core 的共享 lib 目录等，以及 SolrCloud 的相关配置。附带的还有 Solr 日志配置。在 Solr4.4 版本之前，还包括 Core 的定义配置即 Core 需要事先在 solr.xml 中通过 <core> 元素进行定义，版本升级后，Solr 加入了 Core 自动发现机制，<solr> 元素下的 <cores>、<core> 配置元素就被移除了。下面是 solr.xml 的一个完整配置示例：

```
<solr>
  <str name="adminHandler">${adminHandler:org.apache.solr.handler.admin.CoreAdmin-
Handler}</str>
  <int name="coreLoadThreads">${coreLoadThreads:3}</int>
```

```

<str name="coreRootDirectory">${coreRootDirectory:}</str><!-- usually solrHome -->
<str name="managementPath">${managementPath:}</str>
<str name="sharedLib">${sharedLib:}</str>
<str name="shareSchema">${shareSchema:false}</str>
<int name="transientCacheSize">${transientCacheSize:Integer.MAX_VALUE}</int><!--
ignored unless cores are defined with transient=true -->

<solrcloud>
<int name="distribUpdateConnTimeout">${distribUpdTimeout:}</int>
<int name="distribUpdateSoTimeout">${distribUpdateTimeout:}</int>
<int name="leaderVoteWait">${leaderVoteWait:}</int>
<str name="host">${host:}</str>
<str name="hostContext">${hostContext:solr}</str>
<int name="hostPort">${jetty.port:8983}</int>
<int name="zkClientTimeout">${zkClientTimeout:15000}</int>
<str name="zkHost">${zkHost:}</str>
<bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
</solrcloud>
<logging>
<str name="class">${loggingClass:}</str>
<str name="enabled">${loggingEnabled:}</str>
<watcher>
<int name="size">${loggingSize:}</int>
<int name="threshold">${loggingThreshold:}</int>
</watcher>
</logging>

<shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
<int name="socketTimeout">${socketTimeout:}</int>
<int name="connTimeout">${socketTimeout:}</int>
</shardHandlerFactory>
</solr>

```

solr.xml 配置文件支持使用变量，格式为：\${属性名称:默认值}，后面的默认值是可选的，表示当根据属性名称获取属性值发现找不到时会采用冒号后面配置的默认值，当用户没有配置默认值也没有关系，Solr 程序内部会设置默认值。注意，Solr 程序内部设置的默认值可能会随着版本的升级会发生变化，所以实际各个属性默认值需要以实际版本的源码为准。冒号前的值有两种设置途径：

- 1) 通过 JVM 的 -D 参数设置，比如 -DpropertyName=value;
- 2) 通过 core.properties 属性配置文件设置。properties 属性配置文件我想大家都不陌生，直接在文件里通过 key=value 这种键值对方式配置你的参数，每个键值对独占一行。属性配置文件与需要应用属性配置文件里参数的配置文件（比如 solr.xml、solrconfig.xml、data-config.xml 都可以通过 properties 属性配置文件来获取参数）放置在同级目录即可。如果你想要自定义 core.properties 配置文件的加载路径，那么你可以通过 -Dsystem.properties=conf\solr\core.properties 系统参数来自定义。

下面我们来讲解 Core 自动发现机制。

当 Solr 启动时，会在 SOLR\_HOME（即 Solr 的配置主目录，同时默认也是 Solr Core 的宿主目录）目录下递归查找名称为 core.properties 配置文件，然后根据 core.properties 中的配置去加载 Core，也就是说 core.properties 配置文件只需要放置下 SOLR\_HOME 目录下即可，并没有要求它必须要放置在每个 Core 的根目录下，一般一个 Core 对应一个 core.properties 配置文件。core.properties 中支持如下配置参数：

- ❑ name: core 的名称，必需参数；
- ❑ config: 用于指定 solrconfig.xml 配置文件的文件名，默认值是 solrconfig.xml；
- ❑ dataDir: 配置 core 的数据目录，该目录下主要存放索引数据以及事务日志，默认为 data，可选参数；
- ❑ ulogDir: 用于配置事务日志文件的存放目录，默认是存放在 dataDir 的 tlog 目录下，你可以将事务日志目录设置为跟索引数据目录不在同一个硬盘，这样可以减少磁盘竞争提升 I/O 性能。默认这两个目录都处在 dataDir 目录下，会存在 I/O 竞争；
- ❑ schema: 用于配置 Core 的 schema.xml 文件名称，默认是 schema.xml；
- ❑ shard: 配置分片 ID，用于 Solr Cloud 模式下；
- ❑ collection: 配置当前 Core 属于哪个 collection，用于 Solr Cloud 模式下；
- ❑ roles: SolrCloud 的 role 定义，用于 Solr Cloud 模式下；
- ❑ properties: 用于配置 core.properties 文件的加载路径，已经不推荐使用；
- ❑ loadOnStartup: 是否 Solr 启动时就加载当前 Core 并创建一个新的 IndexSearcher 实例；
- ❑ transient: 用来设置当 Solr 的 transient-cacheSize 阈值达到限制值的时候，是否自动卸载当前 core；
- ❑ coreNodeName: 配置 Core 节点名称，用于 Solr Cloud 模式下。

Solr 启动完成后，Solr 不再监视 Core 根目录的任何变化，当根据 Core 的 Http 接口创建一个新 Core 时，若 Core 根目录的 core.properties 配置文件不存在，Solr 会自动创建一个 core.properties 文件，并将当前 Core 的核心信息写入 core.properties，写入内容大致如下所示：

```
name=core1
config=solrconfig.xml
schema=schema.xml
dataDir=data
```

由于 Solr 必须根据 core.properties 配置文件来自动发现 Core，也就意味着，如果某个 Core 的根目录下没有提供 core.properties，那么这个 Core 将会被忽略，并且无法被自动发现和加载。

## 3.2 solrconfig.xml 配置详解

solrconfig.xml 配置文件中包含了很多 Solr 自身配置相关的参数，solrconfig.xml 配置文

件示例可以从 Solr 的解压目录下找到。

solrconfig.xml 中的配置项主要分以下几大块：

- 依赖的 Lucene 版本配置，这决定了你创建的 Lucene 索引结构，因为 Lucene 各版本之间的索引结构并不是完全兼容的，这个需要引起你的注意；
- 索引创建相关的配置，如索引目录，IndexWriterConfig 类中的相关配置（它决定了你的索引创建性能）；
- solrconfig.xml 中依赖的外部 jar 包加载路径配置；
- JMX 相关配置；
- 缓存相关配置，缓存包括过滤器缓存，查询结果集缓存，Document 缓存，以及自定义缓存等；
- updateHandler 配置即索引更新操作相关配置；
- RequestHandler 相关配置，即接收客户端 HTTP 请求的处理类配置；
- 查询组件配置如 HightLight, SpellChecker 等等；
- ResponseWriter 配置即响应数据转换器相关配置，决定了响应数据是以什么样格式返回给客户端的；
- 自定义 ValueSourceParser 配置，用来干预 Document 的权重、评分，排序。

下面是 solrconfig.xml 的配置详解：

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
<!--
在下面的所有配置中，类名之前带有 solr. 前缀的都是别名，solr. 前缀会触发 solr
去扫描 org.apache.solr.(search|update|request|core|analysis) 包，你也可以
为你的自定义插件类指定一个完整的包路径。
-->
<!--
配置 Solr 依赖的 Lucene 版本，一般建议依赖最新发布版本的 Lucene，因为最新版本一般解决了
大量 BUG 并且性能有所提升。如果你修改了 Lucene 版本，强烈建议你的索引数据也重建，因为 Lucene
各版本之间的索引结构不一定完全兼容，这需要你稍作测试。
-->
<.luceneMatchVersion>5.3.1</luceneMatchVersion>
<!-- <lib/>
lib 配置用于告诉 Solr 如何加载在 solrconfig.xml 或者 schema.xml 中定义的插件依赖的 jar 包
（比如 Analyzer 分词器、RequestHandler 请求处理器等等），lib 元素的 dir 属性值是一个相对
于当前 Core 根目录的相对路径，该目录下的所有 jar 包都将被加载。<lib> 元素可以配置多个，类
加载器会严格按照 <lib> 元素定义的顺序从上至下加载，因此低依赖的 jar 包应该优先定义在最前
面，<lib> 元素的 regex 属性支持正则表达式来模糊配置文件名，不符合正则表达式规则的文件将会被
忽略。
-->
<lib dir="${solr.install.dir:../../../../}/contrib/extraction/lib" regex=".*\.jar" />
<lib dir="${solr.install.dir:../../../../}/dist/"
regex="solr-cell-.*\.jar" />
<lib dir="${solr.install.dir:../../../../}/contrib/clustering/lib/" regex=".*\.jar" />
```



```

<lib dir="${solr.install.dir:../../../../}/dist/" regex="solr-clustering-\.d.*\.jar" />
<lib dir="${solr.install.dir:../../../../}/contrib/langid/lib/" regex=".*\.jar" />
<lib dir="${solr.install.dir:../../../../}/dist/" regex="solr-langid-\.d.*\.jar" />
<lib dir="${solr.install.dir:../../../../}/contrib/velocity/lib" regex=".*\.jar" />
<!-- 也可以配置 path 属性来加载一个指定的 jar 或者 properties 属性配置文件 -->
<lib path="${solr.install.dir:../../../../}/example/files/browse-resources"/>
<lib dir="${solr.install.dir:../../../../}/dist/" regex="solr-velocity-\.d.*\.jar" />
<!--

```

配置 Core 的数据目录，默认为 SOLR\_HOME 下的 data 目录。

如果使用了主从复制，那么还需要跟主从复制配置保持一致。

```
-->
```

```
<dataDir>${solr.data.dir:</dataDir>
```

```
<!--
```

DirectoryFactory 用于创建索引。

solr.StandardDirectoryFactory 是以文件系统为基础，尝试为当前 JVM 平台选择最好的实现

solr.NRTCachingDirectoryFactory 是默认实现，它包装了 StandardDirectoryFactory 并在内存中缓存了一些索引文件为了提升近实时搜索性能。

solr.MMapDirectoryFactory 是一个基于虚拟内存映射技术实现的 DirectoryFactory，它允许 Lucene 或 Solr 直接访问 I/O 缓存。如果不需要近实时搜索功能，这个工厂实现是个不错的选择。solr.NIOFSDirectoryFactory 是基于 NIO 实现的 DirectoryFactory，适用于多线程环境，但是不适用于 windows 平台。

solr.SimpleFSDirectoryFactory 是基于文件系统的 DirectoryFactory 简单实现，不适用于多线程高并发大数据场景下。

solr.RAMDirectoryFactory 是基于内存实现的 DirectoryFactory 实现，它并不做文件存储，当系统重启或 Solr Server 崩溃后，索引数据会丢失。而且在主从复制模式下无效。

```
-->
```

```
<directoryFactory name="DirectoryFactory" class="${solr.directoryFactory:solr.NRTCaching
DirectoryFactory}"/>
```

```
<!--
```

CodecFactory 编码工厂类定义了倒排索引的格式，它的默认实现是 SchemaCodecFactory，SchemaCodecFactory 是 Lucene 的官方索引格式，但提供了 schema 钩子来实现 postinglists 和每个 document 的 values 的定制化以及。

注意：大部分可选实现都是实验性质的，不够稳定，因此如果你现在了一个自定义索引格式，建议你将其转换成官方的索引格式，以避免当你版本升级时不必要的索引重建。

```
-->
```

```
<codecFactory class="solr.SchemaCodecFactory"/>
```

```
<!-- ~~~~~
```

```

Index Config - These settings control low-level behavior of indexing
Most example settings here show the default value, but are commented
out, to more easily see where customizations have been made.

```

```
Note: This replaces <indexDefaults> and <mainIndex> from older versions
```

```
~~~~~ -->
```

```
<indexConfig>
```

```
<!-- maxFieldLength 属性配置自 Solr4.0 开始已经被移除，一个可选的替代方案是在你的 schema.xml 中的 fieldType 定义里配置一个 LimitTokenCountFilterFactory，
```

```
比如 <filter class="solr.LimitTokenCountFilterFactory" maxTokenCount="10000"/>
```

```
-->
```

```
<!-- writeLockTimeout: 配置 IndexWriter 的写锁最大超时时间，单位毫秒，默认值 1000 -->
```

```
<!-- <writeLockTimeout>1000</writeLockTimeout> -->
```

```
<!--
```

配置是否启用复合文件，启用复合文件可以减少索引个数，使用更好的文件描述符  
是以牺牲性能为代价的，在 Lucene 中默认是启用的，在 Solr 中自 3.6 版本开始默认值禁用的

```
-->
```

```
<!-- <useCompoundFile>false</useCompoundFile> -->
```

```
<!--
```

ramBufferSizeMB 用于设置 Lucene 需要缓存在创建索引时待添加的 document 以及  
尚未刷新到索引目录的待删除 document 所需的内存缓冲区的大小。单位 MB

maxBufferedDocs 用于设置内存缓冲区缓存的 document 最大个数

如果 ramBufferSizeMB 和 maxBufferedDocs 同时设置了，那么哪个条件先满足就  
触发一次索引刷新写入到索引目录操作

```
-->
```

```
<!-- <ramBufferSizeMB>100</ramBufferSizeMB> -->
```

```
<!-- <maxBufferedDocs>1000</maxBufferedDocs> -->
```

```
<!--
```

配置索引的段文件合并策略

自 Solr/Lucene 3.3 起，默认实现是 TieredMergePolicy

自 Lucene 2.3 起，默认实现是 LogByteSizeMergePolicy

甚至更老版本的 Lucene 中使用的是 LogDocMergePolicy 实现

```
-->
```

```
<mergePolicyFactory class="solr.TieredMergePolicyFactory">
```

```
<!-- 设置一次循环最多合并多少个段文件 -->
```

```
<int name="maxMergeAtOnce">10</int>
```

```
<!-- 设置每层段文件允许的个数，超过限制会触发段文件合并，  
同时也是内存 buffer 递减的等比数列的公比
```

```
-->
```

```
<int name="segmentsPerTier">10</int>
```

```
</mergePolicyFactory>
```

```
-->
```

```
<!-- 合并因子 -->
```

```
<!--
```

IndexWriter 的 mergeFactory 允许你来控制索引在写入磁盘之前内存中能缓存的 document 数量，以及合并多个段文件的频率。默认这个值为 10。当往内存中存储了 10 个 document，此时 Lucene 还没有把单个段文件写入磁盘，mergeFactor 值等于 10 也意味着当硬盘上的段文件数量达到 10，lucene 将会把这 10 个段文件合并到一个段文件中。例如：如果你把 mergeFactor 设置为 10，当你往索引中添加 10 个 document，一个段文件将会在硬盘上被创建，当第 10 个段文件被添加时，这 10 个段文件就会被合并到 1 个段文件，此时这个段文件中有 100 个 document，当 10 个这样的包含了 100 个 document 的段文件被添加时，他们又会被合并到一个新的段文件中，而此时这个段文件包含 1000 个 document，以此类推。所以，在任何时候，在索引中不存在超过 9 个段文件。每个被合并的段文件包含的 document 个数都是 10，但这样有点小问题，我们还必须设置一个 maxMergeDocs 变量，当合并段文件的时候，lucene 必须确保没有哪个段文件超过 maxMergeDocs 变量规定的最大 document 数量。设置 maxMergeDocs 的目的是为了防止单个段文件中包含的 document 数量过大，假定你把 maxMergeDocs 设置为 1000，当你创建第 10 个包含 1000 个 document 段文件的时候，这时并不会触发段文件合并（如果没有设置 maxMergeDocs 为 100 的话，按理来说，这 10 个包含了 1000 个 document 的段文件将会被合并到一个包含了 10000 个 document 的段文件当中，但 maxMergeDocs 限制了单个段文件中最多包含 1000 个 document，所以此时并不会触发段合并操作）。影响段合并还有一些其他参数，比如：

mergeFactor: 当大小几乎相当的段的数量达到此值的时候，开始合并。

minMergeSize: 所有大小小于此值的段，都被认为是大小几乎相当，一同参与合并。

maxMergeSize: 当一个段的大小大于此值的时候, 就不再参与合并。

maxMergeDocs: 当一个段包含的文档数大于此值的时候, 就不再参与合并。

段文件合并分两个步骤:

1. 首先筛选出哪些段需要合并, 这一步由 MergePolicy 合并策略类来决定
2. 然后就是真正的段合并过程了, 这一步是交给 MergeScheduler 来完成的, MergeScheduler 类主要做两件事:
  - A. 对存储域, 项向量, 标准化因子即 norms 等信息进行合并
  - B. 对倒排索引信息进行合并

```
-->
<mergeFactor>10</mergeFactor>
<!--
配置段文件合并调度器实现
    ConcurrentMergeScheduler (Lucene2.3 里是默认实现) 会使用独立线程在
后台执行段文件合并操作
    SerialMergeScheduler (Lucene2.2 里是默认实现) 即串行合并调度器, 性能
比 ConcurrentMergeScheduler (并行合并调度器) 性能要差, 已经不建议使用了
-->
<!--
<mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler"/>
-->
```

用于配置 Lucene 的 LockFactory 实现  
提供了以下可选实现:

```
single = SingleInstanceLockFactory
```

当索引是只读的或不存在另外一个程序试图修改索引时建议使用这种实现

```
native = NativeFSLockFactory
```

使用操作系统本地的锁, 当在同一个 JVM 实例里部署了多个 Solr webapp 并  
试图共享同一个索引数据时, 不建议使用 NativeFSLockFactory

```
simple = SimpleFSLockFactory
```

使用简单的文件来实现锁, 通过在硬盘上创建 write.lock 锁文件实现

Solr3.6 以及更老的版本中, native 是默认实现, 其他版本默认实现是 simple

关于各 LockFactory 之间的细微差别详情请访问

<http://wiki.apache.org/lucene-java/AvailableLockFactories>

```
-->
<lockType>${solr.lock.type:native}</lockType>
<!--
```

unlockOnStartup 告知 Solr 忽略在多线程环境中用来保护索引的锁定机制。在某些情况下, 索引可能会  
由于不正确的关机或其他错误而一直处于锁定, 这就影响了索引的添加和更新。将其设置为 true 可以禁  
用启动锁定, 进而允许进行添加和更新。

```
-->
<unlockOnStartup>false</unlockOnStartup>
<!-- Lucene 加载 term 到内存中的时间间隔 -->
<termIndexInterval>128</termIndexInterval>
<!-- 重新打开 IndexReader, 替代先关闭 - 再打开 -->
<reopenReaders>true</reopenReaders>
<!--
```

配置删除提交策略

自定义的删除提交策略可以在这里配置, 该类必须实现

```
org.apache.lucene.index.IndexDeletionPolicy
```

Solr 的默认实现 IndexDeletionPolicy 支持待删除索引提交点, 提交点包含如下信息:

提交文档数量

提交点的年龄（每提交一次，被缓存下来的提交点 age+1）

经过优化的状态

最新提交点会一直无条件被缓存下来

```
-->
<!--
<deletionPolicy class="solr.SolrDeletionPolicy">
-->
<!-- 缓存的提交点最大个数 -->
<!-- <str name="maxCommitsToKeep">1</str> -->
<!-- 缓存的经过优化的提交点最大个数 -->
<!-- <str name="maxOptimizedCommitsToKeep">0</str> -->
<!--
当提交点年龄达到限定的最大值，那么该提交点会被删除
提交点最大年龄限制支持 DateMathParser 语法，示例如下
-->
<!--
<str name="maxCommitAge">30MINUTES</str>
<str name="maxCommitAge">1DAY</str>
-->
<!--
</deletionPolicy>
-->
<!-- Lucene Infostream
Lucene 提供了 InfoStream 功能，可以显示索引建立时的详细信息，将
此配置设置为 true 即告诉底层的 Lucene IndexWriter 实例输出 debug 信息
到指定的文件中，配置示例如下所示：
-->
<!-- <infoStream file="INFOSTREAM.txt">false</infoStream> -->
</indexConfig>
<!--
用于启用 JMX，启用 JMX 只需要找到一个 MBeanServer 即可，
如果你希望通过 JVM 参数来配置 JMX，那么就可以使用这个配置，
如果你想禁止暴露 Solr 的配置信息以及统计信息给 JMX，那么你可以
移除这个配置，详情请访问 http://wiki.apache.org/solr/SolrJmx
-->
<jmx />
<!-- 指定一个 agentId 来连接一个特定的 MBeanServer -->
<!-- <jmx agentId="myAgent" /> -->
<!-- 配置并启动一个新的 MBeanServer -->
<!-- <jmx serviceUrl="service:jmx:rmi:///jndi/rmi://localhost:9999/solr"/> -->
<!-- 默认的高性能 update handler 处理器实现 -->
<updateHandler class="solr.DirectUpdateHandler2">
<!--
启用事务日志配置，用于近实时查询、持久性、SolrCloud 副本恢复，事务日志
会随着未提交到索引目录的更新增多而变大，所以建议使用自动硬提交。
dir: 配置事务日志文件的存放目录，默认是 solr core 的 data 目录
numVersionBuckets: 用于记录最大版本号轨迹的桶的容量大小，版本号桶用于检测重排序更新，当你
对大量文档创建索引时，可以增大此值以减小同步访问版本号桶的开销。对于每个 Solr Core，需要分配 8
bytes (long) * numVersionBuckets 的堆内存给版本号桶。
-->
```

```
-->
<updateLog>
<str name="dir">${solr.ulong.dir}</str>
<int name="numVersionBuckets">${solr.ulong.numVersionBuckets:65536}</int>
</updateLog>
<!--
```

关于 AutoCommit (自动提交) 的配置

AutoCommit

在某些提交下自动执行提交。

在添加索引时, 启用了自动提交, 可以考虑使用 commitWithin 参数

maxDocs - 自上一次提交起, 新增的 document 达到限定的最大个数后, 自动触发一次提交

maxTime - 一个文档自添加 (添加默认只是放入内存缓冲区内) 后缓存时间超过限定的最大毫秒数, 就自动触发一次提交

openSearcher - 如果设置为 false, 提交会使最近的索引更新写入到索引目录, 但不会创建一个新的 IndexSearcher 实例。创建一个新的 IndexSearcher 实例使那些最近更新的索引数据立即

可见。反之, 则会创建一个新的 IndexSearcher 实例。

如果你启用了事务日志功能, 那么强烈建议你对自动提交进行排序以限制日志文件的大小。

```
-->
<autoCommit>
<!-- maxTime: 设置多长时间提交一次 -->
<maxTime>15000</maxTime>
<!-- 自动硬提交完成后是否开启一个新的 IndexSearcher 实例 -->
<openSearcher>false</openSearcher>
</autoCommit>
<!--
softAutoCommit 跟 autoCommit 类似, 会自动触发一个软提交, 它只保证更新的索引数据课件当不保证数据同步到硬盘上, 相比硬提交, 它执行速度更快且具有更友好的近实时效果。
```

```
-->
<!--
<autoSoftCommit>
<!-- maxTime: 设置多长时间软提交一次 -->
<maxTime>1000</maxTime>
</autoSoftCommit>
```

-->

```
<!--
与 update 相关的事件监听器配置
各种各样与 IndexWriter 相关的事件会触发监听器作出相应的动作
postCommit - 每一次索引提交或者索引优化时会触发一次 postCommit
postOptimize - 每一次索引优化时会触发一次 postOptimize
```

-->

```
<!--
RunExecutableListener 监听器执行来自类似 postCommit 和 postOptimize 之类的内部命令:
```

exe - 事件触发时需要执行的可执行文件的名称

dir - 当前工作目录, 默认为 . 即当前所在目录

wait - 调用线程需要等待直到可执行文件返回, 默认为 true

args - 传递给应用程序的参数, 默认为空

env - 用于设置系统环境变量

```

-->
<!--
这个示例演示了基于脚本的 replication 模式下 RunExecutableListener 是如何使用的
具体请参阅官方 Wiki: http://wiki.apache.org/solr/CollectionDistribution
-->
<!--
<listener event="postCommit" class="solr.RunExecutableListener">
<str name="exe">solr/bin/snapshooter</str>
<str name="dir">.</str>
<bool name="wait">true</bool>
<arr name="args"><str>arg1</str><str>arg2</str></arr>
<arr name="env"><str>MYVAR=val1</str></arr>
</listener>
-->
</updateHandler>
<!--

```

#### IndexReaderFactory 相关配置

**\*\* 实验性功能 \*\***

请注意：使用一个自定义的 IndexReaderFactory 可能会影响部分其他功能的正常工作。如果问题没有解决的话，IndexReaderFactory 的 API 可能会在毫无提示的情况下被更新，甚至可能会在后续的发布版本中被移除。

**\*\* 自定义的 IndexReaderFactory 可能部分功能无法使用 \*\***

自定义实现 IndexReader 可能会导致不兼容 ReplicationHandler 并且可能导致 replication 不能正常工作，请查阅 See SOLR-1366 了解更多详情。

```

-->
<!--
<indexReaderFactory name="IndexReaderFactory" class="package.class">
<str name="someArg">Some Value</str>
</indexReaderFactory>
-->
<!-- ~~~~~~
Query 篇 - 这些配置用于控制查询时间，比如 cache
~~~~~
-->
<query>
<!--

```

#### Max Boolean Clauses

设置每个 BooleanQuery 最多支持的条件个数，如果超出会抛出异常。

**\*\* 警告 \*\***

这个配置选项实际上会修改 Lucene 的全局属性，这会影响到所有的 Solr Core。

如果多个 solrconfig.xml 在这个配置项不一致，那么这个配置项的值在任何时刻都会以最后初始化的那个 Core 的配置为准。

```

-->
<maxBooleanClauses>1024</maxBooleanClauses>
<!-- Solr 内部的 Query 缓存
在 Solr 里有两个可用的缓存实现

```

LRUCache: 基于同步的 LinkedHashMap 实现的

FastLRUCache: 基于 ConcurrentHashMap 实现的

FastLRUCache 在单线程下拥有更快的读速度更慢的写速度，因此，通常情况下，当缓存命中率大于 75% 或者多 CPU 的其他场景下，我们认为 FastLRUCache 要快于



## LRUCache

--&gt;

## &lt;!-- Filter 缓存配置

这个缓存被 SolrIndexSearcher 用来过滤生成 DocSets, 当一个 IndexSearcher 实例被打开时, 它的缓存会被填充或者从一个旧的 IndexSearcher 实例的缓存里提前预热, autowarmCount 表示填充的缓存项的个数, 对于 LRUCache 来说, 自动预热的缓存项将会是绝大部分的最近访问的缓存项。

参数:

class - SolrCache 的实现类: LRUCache 或者 FastLRUCache  
size - 缓存中实体的最大数目  
initialSize - 初始化缓存中实体的最大容量  
autowarmCount - 从旧 IndexSearcher 实例的缓存中自动预热填充的实体个数

--&gt;

```
<filterCache class="solr.FastLRUCache"size="512"
  initialSize="512"
  autowarmCount="0"/>
```

&lt;!--

Query Result Cache 查询结果缓存  
IndexSearcher 的查询结果集缓存是 document id 集合 (DocList)  
docList 在查询、排序、document 的范围查询会使用以提升性能  
LRUCache 额外支持的参数: maxRamMB - 用于配置当前缓存在内存中允许占用的最大空间, 单位: MB

--&gt;

```
<queryResultCache class="solr.LRUCache"size="512"
  initialSize="512"autowarmCount="0"/>
```

&lt;!-- Document Cache (Document 缓存)

用于缓存 Lucene 的 Document 对象, 因为 Lucene 内部的 document ids 是瞬态的, 所以 Document cache 不能被自动预热。

--&gt;

```
<documentCache class="solr.LRUCache" size="512"
  initialSize="512"autowarmCount="0"/>
```

&lt;!-- Field Value Cache (域值缓存)

此 Cache 用于缓存域值, 以便于能快速的通过文档 ID 访问到域值域值缓存默认是开启的

--&gt;

&lt;!--

```
<fieldValueCache class="solr.FastLRUCache"size="512"
  autowarmCount="128"showItems="32" />
```

--&gt;

&lt;!-- Custom Cache (自定义缓存)

自定义缓存可以通过 SolrIndexSearcher.getCache(), cacheLookup() 来访问, 通过 cacheInsert() 来插入缓存, 设计自定义缓存是为了启用用户或者应用程序级别数据的简单缓存。当你希望使用自动预热功能, 你需要通过 regenerator 参数来指定自定义的 CacheRegenerator 实现, 你自己的 Regenerator 需要实现 Solr 的 CacheRegenerator 接口。

--&gt;

&lt;!--

```
<cache name="myUserCache"
  class="solr.LRUCache"size="4096"
  initialSize="1024"autowarmCount="1024"
  regenerator="com.mycompany.MyRegenerator"
```

/&gt;

--&gt;

<!-- Lazy Field Loading (域延迟加载)

如果设置为 true, 没有被请求的存储域会被延迟加载, 在不需要加载所有存储域的用户场景下, 特别是被跳过的域是一些大压缩文本时, 这个配置会带来非常大的速度提升。

--&gt;

<enableLazyFieldLoading>true</enableLazyFieldLoading>

<!-- Use Filter For Sorted Query (为带排序的 Query 启用 Filter 过滤器)

试图使用 filter 去满足一个查询的合理优化, 如果被请求的排序操作不需要

返回文档评分, 那么然后 filter 会检查 filterCache 来匹配 query 查询, 如果

在 filterCache 中找到, 那么这个排序操作就会应用这个找到的缓存项。

说简单点就是: 当你的 Query 没有使用 score 进行排序时, 是否使用 filter 来

替代 Query. 对于大部分场景, 这个配置项是没有用的, 除非你频繁使用不同的排序规则来频繁的重复执行相同的查询请求并且不要求返回文档评分 score, 这个配置才有用。

--&gt;

<!--

<useFilterForSortedQuery>true</useFilterForSortedQuery>

--&gt;

<!-- Result Window Size

使用查询结果集缓存的一个优化。当执行一个查询请求时, 会收集到返回的 document

的 id 集合, 举个例子, 如果一个特定的查询请求匹配 10-19 的 document, 此时 queryWindowSize 等于

50, 然后 0-49 的 document 会被收集并缓存, 后续在这个范围内的查询请求会直接命中这个缓存。query-

ResultWindowSize 参数表示在一次查询中缓存的 documentid 最大个数, 一般与 queryResultCache 搭配使用

--&gt;

<queryResultWindowSize>20</queryResultWindowSize>

<!-- 在查询结果集缓存中能够缓存的 document 最大个数 -->

<queryResultMaxDocsCached>200</queryResultMaxDocsCached>

<!-- Query Related Event Listeners (查询相关的事件监听器配置)

各种各样的 IndexSearcher 相关的事件可以触发监听器去作出相应的动作

newSearcher - 当一个新的 IndexSearcher 实例正在被初始化并且当前

IndexSearcher 实例正在处理查询请求时会触发, 它可以用于大部分缓存场景下来避免耗时很长的请求。

newSearcher 用于当一个新的 IndexSearcher 实例被创建时, 除了从旧 IndexSearcher 实例自动预热一部分缓存之外, 你还可以显式的指定一个查询来对缓存进行预热。

firstSearcher- 当一个新的 IndexSearcher 实例正在被初始化并且当前没有旧的 IndexSearcher 实例用于新的 IndexSearcher 实例进行缓存自动预热, 此时你需要显式的指定一个查询来自动预热缓存。这个 firstSearcher 主要用于配置 Solr 刚启动时执行什么查询并放入缓存

--&gt;

<listener event="newSearcher" class="solr.QuerySenderListener">

<arr name="queries">

<!--

<lst><str name="q">solr</str><str name="sort">price asc</str></lst>

<lst><str name="q">rocks</str><str name="sort">weight asc</str></lst>

-->

</arr>

</listener>

<listener event="firstSearcher" class="solr.QuerySenderListener">

<arr name="queries">

```

<!--
<lst>
<str name="q">static firstSearcher warming in solrconfig.xml</str>
</lst>
-->
</arr>
</listener>

<!-- Use Cold Searcher(使用冷查询)
如果一个查询请求发起了,但 IndexSearcher 实例还没有注册,然后会立即将
正在进行缓存预热的 IndexSearcher 实例进行注册,如果设置为 false,那么
所有的查询请求会被阻塞直到 firstSearcher 实例完成了缓存预热。
-->
<useColdSearcher>false</useColdSearcher>
<!-- Max Warming Searchers(预热 IndexSearcher 实例最大个数)
在后台并发进行缓存预热的 IndexSearcher 实例最大个数,如果超出了限定的
最大值会抛出异常。
对于只读的从节点,这个配置的值推荐设置为 1-2
对于正在预热缓存的主节点,这个配置的值可以设置稍微大点
-->
<maxWarmingSearchers>2</maxWarmingSearchers>
</query>
<!-- Request Dispatcher(请求分发器配置)
这部分包含了当处理 SolrCore 的请求时 SolrDispatchFilter 是如何运行的说明
handleSelect 是一个历史遗留的配置项,它影响请求的行为比如 /select?qt=XXX
handleSelect="true" 会促使 SolrDispatchFilter 处理请求并通过 qt 参数分发查询
给指定的 handler,假设 "/select" 还没有注册, "/select" 请求将会被忽略,并
返回一个 404 页面,除非你显式的注册一个名称为 "/select" 的 handler。
对于新手来说,handleSelect="true" 是不推荐的,允许 handleSelect="true" 是为了
保持与老版本的兼容性。
-->
<requestDispatcher handleSelect="false" >
<!-- Request Parsing(请求语解析器配置)
这些配置指示了 Solr 该如何解析请求以及请求的 ContentStreams 有哪些限制约束
enableRemoteStreaming - 表示为指定的远程 streams 启用 stream.file 和 stream.
url 参数
multipartUploadLimitInKB - 指定了 Solr 允许请求中上传的文件最大体积,单位: KB
formdataUploadLimitInKB - 指定 POST 请求提交的表单数据最大大小,单位: KB
addHttpRequestToContext - 如果设置为 true,即告诉 requestParsers 将原始的 Http-
ServletRequest 对象添加到
SolrQueryRequest 的上下文对象的 Map 中,其中 key 为 "httpRequest" 这个配置对于 Solr
的内置组件无效,当你想要开发自己的自定义插件时,这个配置可能会有用。
*** 警告 ***
下面的这些配置委托 Solr 去提取远程文件,当你将 enableRemoteStreaming 设置为
true,你需要确保你的系统有一定的身份认证机制来保证系统安全性(意思就是你需要采取
一些措施防止用户上传恶意文件攻击你的系统)
-->
<requestParsers enableRemoteStreaming="true"
multipartUploadLimitInKB="2048000"
formdataUploadLimitInKB="2048"

```

```

        addHttpRequestToContext="false"/>
<!-- HTTP Caching (Http 缓存相关的配置)
设置域 Http 缓存相关的参数 (针对缓存代理和客户端)
下面的配置项告诉 Solr 不要在请求头信息里输出任何 http cache 信息
304 状态码即表示 Http Server 告诉 Client, 之前输出给你的页面没有更新, 你继续
使用上一次返回的页面吧, 那么 never304 就是取反, 即禁用 Http 缓存
-->
<httpCaching never304="true" />
<!--
如果你在 httpCaching 元素里配置了 <cacheControl>, 那么就会在请求头信息里自动
生成一个 Cache-Control, 默认不生成。即便你已经配置了 never304="true",
<cacheControl> 配置依然有效。
-->
<!--
<httpCaching never304="true" >
<cacheControl>max-age=30, public</cacheControl>
</httpCaching>
-->
<!--
想要启用自动在响应头信息里生成 http cache header, 那么你需要设置 never304 为 false。
这个配置会促使 Solr 生成 Last-Modified 和 ETag 头信息, 而且还会影响如下头
信息的值:
lastModFrom:
    默认值为 "openTime", 它表示 Last-Modified 的值, 如果你希望这个值与物理索引文件的
    最后一次修改时间相对应, 那么你可以将其修改为 dirLastMod。
etagSeed:
    改变这个值可以有助于强制用户重新取得内容, 即使索引数据没有更新, 如果你配置了 never304=
    "true", 那么 lastModifiedFrom 和 etagSeed 配置都将其失效。
-->
<!--
<httpCaching lastModifiedFrom="openTime"
                etagSeed="Solr">
<cacheControl>max-age=30, public</cacheControl>
</httpCaching>
-->
</requestDispatcher>
<!-- Request Handlers (request handler 相关配置)
客户端可以通过带有 "qt" 参数的 "/select/ url" 请求, 也可以通过在 solrconfig.xml
配置的方式来决定要访问的 SolrRequestHandler。
Solr 是通过以下逻辑去选择一个 handler 并处理请求的:
查找 name 属性与请求中的 qt 参数匹配的 handler, 若没找到, 则
查找 "default=true" 的 handler, 若没找到, 则继续
查找 name="standad" 的 handler, 否则使用默认的 StandardRequestHandler 实例
如果你的配置文件 solrconfig.xml 包含有 name 属性为 "/select", "/update", 或
"/admin", 那么你的程序将不会沿用标准的请求处理过程, 而将会是使用你自己自定义的逻辑。
扩展自己的 Handler
实现一个 SolrRequestHandler 最简单的方法是继承 RequestHandlerBase 类。
如果 request handler 配置了 startup="lazy" 参数, 那么 Solr 启动时它不会被初始化,
只有当使用到它的第一次请求触发时 request handler 才会被初始化。即 request handler
的延迟初始化机制。
-->

```

```

-->
<!--
    SearchHandler
    http://wiki.apache.org/solr/SearchHandler
    SearchHandler 用于处理查询请求，SearchHandler 代理了一系列的查询组件，并且它支持
    跨分片的分布式查询
-->

<requestHandler name="/select" class="solr.SearchHandler">
<!--
默认查询参数，这些参数可以在请求时进行覆盖
-->
<lst name="defaults">
<str name="echoParams">explicit</str>
<int name="rows">10</int>
<!-- <str name="df">text</str> -->
</lst>
<!--
除了 defaults 之外，也可以指定 "appends" 参数将多个参数追加到查询请求中
<!--
这个示例中，"fq=instock:true" 参数会在查询时追加 fq 参数
一般不建议配置这个参数，除非你明确知道这个内部机制
-->
<!--
<lst name="appends">
<str name="fq">instock:true</str>
</lst>
-->
<!--
"invariants" 用于配置一些请求参数常量，即用户不能再重置覆盖该配置参数
一般不建议配置这个参数，除非你已经了解了这个配置的含义
-->
<!--
<lst name="invariants">
<str name="facet.field">cat</str>
<str name="facet.field">manu_exact</str>
<str name="facet.query">price:[* TO 500]</str>
<str name="facet.query">price:[500 TO *]</str>
</lst>
-->

<!--
如果你不希望使用默认的查询组件，你可以覆盖默认的查询组件列表
-->
<!--
<arr name="components">
<str>nameOfCustomComponent1</str>
<str>nameOfCustomComponent2</str>
</arr>
-->
<!-- 配置 HttpShardHandlerFactory 用于分布式查询 -->

```

```
<!--
```

HttpShardHandlerFactory 支持的配置参数如下：

socketTimeout. 默认值 0（使用操作系统默认值）- 等待一个 socket 的最大超时时间，单位：毫秒

connTimeout. 默认值 0（使用操作系统默认值）- 连接一个 socket 的最大超时时间，单位：毫秒

maxConnectionsPerHost. 默认值 20 - 在分布式查询中每个分片的最大连接数

corePoolSize. 默认值 0 - 在分布式查询中线程池的初始化大小

maximumPoolSize. 默认值 Integer.MAX\_VALUE - 在分布式查询中线程池的最大容量

maxThreadIdleTime. 默认值 5，设置线程在被回收之前空闲的最大时间，单位：秒

sizeOfQueue. 默认值 1 - 是否使用阻塞队列替代 direct hand off. 这个可能很难理解，其实注重高吞吐量的系统会更希望配置为 direct hand off（即 -1），而注重低延迟性的系统会更希望配置成一个合理大小的队列来处理各种请求。

fairnessPolicy. 默认值 false - 表示是否启用 JVM 中队列的公平性策略

如果启用，那么分布式查询会以先进先出的方式来处理请求但会牺牲一部分系统吞吐量作为代价

如果禁用，那么吞吐量会优于延迟性

```
-->
```

```
<shardHandlerFactory class="HttpShardHandlerFactory">
```

```
<int name="socketTimeOut">1000</int>
```

```
<int name="connTimeOut">5000</int>
```

```
</shardHandlerFactory>
```

```
</requestHandler>
```

```
<!-- 返回 JSON 的 request handler -->
```

```
<requestHandler name="/query" class="solr.SearchHandler">
```

```
<lst name="defaults">
```

```
<str name="echoParams">explicit</str>
```

```
<str name="wt">json</str>
```

```
<str name="indent">true</str>
```

```
</lst>
```

```
</requestHandler>
```

```
<!--useParams 的值定义在 params.json 文件中-->
```

```
<requestHandler name="/browse" class="solr.SearchHandler" useParams="query,facets,velocity,browse"/>
```

```
<initParams path="/update/**,/query,/select,/tvrh,/elevate,/spell,/browse">
```

```
<lst name="defaults">
```

```
<str name="df">_text_</str>
```

```
</lst>
```

```
</initParams>
```

```
<initParams path="/update/**">
```

```
<lst name="defaults">
```

```
<str name="update.chain">files-update-processor</str>
```

```
</lst>
```

```
</initParams>
```

```
<!-- Solr Cell Update Request Handler
```



用于处理富文本文件（比如 Word, PDF）并创建索引

<http://wiki.apache.org/solr/ExtractingRequestHandler>

-->

```
<requestHandler name="/update/extract"
    startup="lazy"
    class="solr.extraction.ExtractingRequestHandler" >
  <lst name="defaults">
    <str name="xpath">/xhtml:html/xhtml:body/descendant::node()</str>
    <str name="capture">content</str>
    <str name="fmap.meta">attr_meta_</str>
    <str name="uprefix">attr_</str>
    <str name="lowernames">true</str>
  </lst>
</requestHandler>
```

<!-- Field Analysis Request Handler(域分析请求处理器配置)

提供了在索引时和查询时对指定的域类型或域进行分析并输出分析结果

请求参数如下：

analysis.fieldname - 需要分析的域名称

analysis.fieldtype - 需要分析的域类型

analysis.fieldvalue - 需要分析的域的域值

q (or analysis.q) - 查询时需要分析的文本内容

analysis.showmatch (true|false) - 设置为 true, 对域值进行分析生成的 token 会被标记为 "matched"

-->

```
<requestHandler name="/analysis/field"
    startup="lazy"
    class="solr.FieldAnalysisRequestHandler" />
```

<!-- Document Analysis Handler(文档分析处理器配置)

<http://wiki.apache.org/solr/AnalysisRequestHandler>

它提供了一个针对指定的 document 进行故障分析, 它期望的 document 格式为:

```
<docs>
<doc>
<field name="id">1</field>
<field name="name">The Name</field>
<field name="text">The Text Value</field>
</doc>
<doc>...</doc>
<doc>...</doc>
```

...

</docs>

注意: 每个 document 必须包含一个主键 field

类似 FieldAnalysisRequestHandler, 这个处理器也提供了查询分析功能

analysis.query 参数表示对那个查询进行分析, q 参数表示查询关键字, 同时也支持 analysis.showmatch 参数, 如果 analysis.showmatch=true, 那么匹配指定 query 查询的 token 都会被标记为 "matched"

-->

```
<requestHandler name="/analysis/document"
```

```

        class="solr.DocumentAnalysisRequestHandler"
        startup="lazy" />

<!-- Echo the request contents back to the client -->
<requestHandler name="/debug/dump" class="solr.DumpRequestHandler" >
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <str name="echoHandler">true</str>
  </lst>
</requestHandler>

```

<!-- Search Components ( 查询组件配置 )

查询组件会被注册为 Solr Core, 并被 SearchHandler 所使用。

默认有以下查询组件可用:

```

<searchComponent name="query"          class="solr.QueryComponent" />
<searchComponent name="facet"          class="solr.FacetComponent" />
<searchComponent name="mlt"            class="solr.MoreLikeThisComponent" />
<searchComponent name="highlight"      class="solr.HighlightComponent" />
<searchComponent name="stats"          class="solr.StatsComponent" />
<searchComponent name="debug"          class="solr.DebugComponent" />

```

查询组件在 request handler 中默认配置示例如下:

```

<arr name="components">
  <str>query</str>
  <str>facet</str>
  <str>mlt</str>
  <str>highlight</str>
  <str>stats</str>
  <str>debug</str>
</arr>

```

如果你注册一个查询组件, 不能使用默认内置的组件名称, 否则会覆盖默认的查询组件, 如果需要将某个组件注册到 'standard' components 之前, 你可以使用 first-components, 配置示例如下:

```

<arr name="first-components">
  <str>myFirstComponentName</str>
</arr>
同理还有 last-components
<arr name="last-components">
  <str>myLastComponentName</str>
</arr>

```

注意:

名称为 debug 的查询组件会一直在 last-components 之后执行, 意思就是 last-components 虽然名称上看起来是最后一个被执行的查询组件, 其实无论如何, debug 查询组件都永远是最后一个被执行的查询组件。

```
-->
```

<!-- Spell Check ( 拼写检查组件配置 )

拼写检查组件会返回可选的拼写建议列表

具体请查阅官方 Wiki:

```

    http://wiki.apache.org/solr/SpellCheckComponent [3]
-->
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">

<str name="queryAnalyzerFieldType">text_general</str>

<!-- 可以通过这个组件声明多个 Spell Checker -->

<!-- 根据主索引的某个 field 构建一个 spellchecker -->
<lst name="spellchecker">
  <str name="name">default</str>
  <str name="field">text</str>
  <str name="classname">solr.DirectSolrSpellChecker</str>
  <!-- 配置 spellchecker 的编辑距算法实现，内部默认是基于 levenshtein 算法实现 -->
  <str name="distanceMeasure">internal</str>
  <!-- 合法拼写建议的最小精确度 -->
  <float name="accuracy">0.5</float>
  <!-- 枚举 term 时的最大编辑距，[1,2] 之间 -->
  <int name="maxEdits">2</int>
  <!-- 枚举 term 时的最小前缀 -->
  <int name="minPrefix">1</int>
  <!-- 每个结果的最大检查数量 -->
  <int name="maxInspections">5</int>
  <!-- 查询字符最小长度，小于此长度将不会启用拼写检查 -->
  <int name="minQueryLength">4</int>
  <!-- 最大查询频率，大于这个值就不会启用拼写检查，Solr 认为频繁被作为搜索关键词的
  热词不需要拼写检查 -->
  <float name="maxQueryFrequency">0.01</float>
  <!-- token 在 document 中出现的频率阈值，Solr 认为出现频率低的 token 更需要拼写检查，
  这个配置不推荐配置，保持默认即可 -->
  <float name="thresholdTokenFrequency">.01</float>
-->
</lst>

<!-- Solr 内置的 WordBreakSolrSpellChecker 配置，支持单词的分裂和合并 -->
<lst name="spellchecker">
  <str name="name">wordbreak</str>
  <str name="classname">solr.WordBreakSolrSpellChecker</str>
  <str name="field">name</str>
  <str name="combineWords">true</str>
  <str name="breakWords">true</str>
  <int name="maxChanges">10</int>
</lst>

<!-- 基于另一种编辑距算法实现的 spellchecker -->
<!--
<lst name="spellchecker">
  <str name="name">jarowinkler</str>
  <str name="field">spell</str>

```

```

<str name="classname">solr.DirectSolrSpellChecker</str>
<str name="distanceMeasure">
    org.apache.lucene.search.spell.JaroWinklerDistance
</str>
</lst>
-->

```

<!-- 基于一种可选的比较器实现的 spellchecker  
比较器可以是以下其中任意一种：

1. score (default)
2. freq ( 首先按频率比较, 再按评分比较 )
3. 类的完整包路径, 用于自定义的比较器实现

```

-->
<!--
<lst name="spellchecker">
<str name="name">freq</str>
<str name="field">lowerfilt</str>
<str name="classname">solr.DirectSolrSpellChecker</str>
<str name="comparatorClass">freq</str>
-->

```

<!-- 基于读取拼写检查配置文件实现的 spellchecker -->

```

<!--
<lst name="spellchecker">
<str name="classname">solr.FileBasedSpellChecker</str>
<str name="name">file</str>
<str name="sourceLocation">spellings.txt</str>
<str name="characterEncoding">UTF-8</str>
<str name="spellcheckIndexDir">spellcheckerFile</str>
</lst>
-->
</searchComponent>

```

<!-- 拼写检查器的 requestHandler 配置

See <http://wiki.apache.org/solr/SpellCheckComponent>

```

-->
<requestHandler name="/spell" class="solr.SearchHandler" startup="lazy">
<lst name="defaults">
<str name="spellcheck.dictionary">default</str>
<str name="spellcheck.dictionary">wordbreak</str>
<str name="spellcheck">on</str>
<str name="spellcheck.extendedResults">true</str>
<str name="spellcheck.count">10</str>
<str name="spellcheck.alternativeTermCount">5</str>
<str name="spellcheck.maxResultsForSuggest">5</str>
<str name="spellcheck.collate">true</str>
<str name="spellcheck.collateExtendedResults">true</str>
<str name="spellcheck.maxCollationTries">10</str>
<str name="spellcheck.maxCollations">5</str>
</lst>

```

```

<arr name="last-components">
<str>spellcheck</str>
</arr>
</requestHandler>

<!-- Term Vector Component (Term 向量查询组件)
      http://wiki.apache.org/solr/TermVectorComponent
-->
<searchComponent name="tvComponent" class="solr.TermVectorComponent"/>

<!-- Term 向量查询处理器 -->
<requestHandler name="/tvrh" class="solr.SearchHandler" startup="lazy">
<lst name="defaults">
<bool name="tv">true</bool>
</lst>
<arr name="last-components">
<str>tvComponent</str>
</arr>
</requestHandler>

<!-- Clustering Component. (Omitted here. See the default Solr example for a typical
configuration.) -->

<!-- Terms Component (Term 查询组件)
      http://wiki.apache.org/solr/TermsComponent
      这个组件会返回 Term 以及 Term 的出现频率
-->
<searchComponent name="terms" class="solr.TermsComponent"/>

<!-- Term 查询组件的请求处理器配置 -->
<requestHandler name="/terms" class="solr.SearchHandler" startup="lazy">
<lst name="defaults">
<bool name="terms">true</bool>
<!-- 是否分布式查询 -->
<bool name="distrib">false</bool>
</lst>
<arr name="components">
<str>terms</str>
</arr>
</requestHandler>

<!-- Query Elevation Component (Solr 的竞价排名查询组件)
      http://wiki.apache.org/solr/QueryElevationComponent
      这个组件提供了可配置性的针对给定的查询直接无视文档的评分而将指定的
      结果排到 Top N 位置即竞价排名
-->
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
<!-- 查询的域类型 -->
<str name="queryFieldType">string</str>

```

```

<!-- 竞价排名配置文件加载路径 -->
<str name="config-file">elevate.xml</str>
</searchComponent>

<!-- 竞价排名组件的请求处理器配置 -->
<requestHandler name="/elevate" class="solr.SearchHandler" startup="lazy">
<lst name="defaults">
<str name="echoParams">explicit</str>
</lst>
<arr name="last-components">
<str>elevator</str>
</arr>
</requestHandler>

<!-- Highlighting Component (高亮组件)
      http://wiki.apache.org/solr/HighlightingParameters
-->
<searchComponent class="solr.HighlightComponent" name="highlight">
<highlighting>
<!-- fragmenter 配置 -->
<fragmenter name="gap"
              default="true"
              class="solr.highlight.GapFragmenter">
<lst name="defaults">
<int name="hl.fragsize">100</int>
</lst>
</fragmenter>

<!-- 基于正则表达式的 fragmenter 配置 -->
-->
<fragmenter name="regex"
              class="solr.highlight.RegexFragmenter">
<lst name="defaults">
<!-- fragsizes 设置小点效果更好 -->
<int name="hl.fragsize">70</int>
<!-- Fragmente 溢出百分比 -->
<float name="hl.regex.slop">0.5</float>
<!-- 句子匹配正则表达式 -->
<str name="hl.regex.pattern">[-\w ,/\n\&quot;&apos;]{20,200}</str>
</lst>
</fragmenter>

<!-- 配置标准的 formatter -->
<formatter name="html"
            default="true"
            class="solr.highlight.HtmlFormatter">
<lst name="defaults">
<str name="hl.simple.pre"><![CDATA[<em>]]></str>
<str name="hl.simple.post"><![CDATA[</em>]]></str>
</lst>

```



```

</formatter>

<!-- 配置标准的 encoder 编码器 -->
<encoder name="html"
        class="solr.highlight.HtmlEncoder"/>

<!-- 配置标准的 fragListBuilder -->
<fragListBuilder name="simple"
        class="solr.highlight.SimpleFragListBuilder"/>

<!-- 配置 SingleFragListBuilder -->
<fragListBuilder name="single"
        class="solr.highlight.SingleFragListBuilder"/>

<!-- 配置加权的 fragListBuilder -->
<fragListBuilder name="weighted"
        default="true"
        class="solr.highlight.WeightedFragListBuilder"/>

<!-- 配置 ScoreOrderFragmentsBuilder -->
<fragmentsBuilder name="default"
        default="true"
        class="solr.highlight.ScoreOrderFragmentsBuilder">

<!--
<lst name="defaults">
<str name="hl.multiValuedSeparatorChar"></str>
</lst>
-->
</fragmentsBuilder>

<!-- 配置多颜色的 FragmentsBuilder -->
<fragmentsBuilder name="colored"
        class="solr.highlight.ScoreOrderFragmentsBuilder">
<lst name="defaults">
<str name="hl.tag.pre"><![CDATA[
<b style="background:yellow">,<b style="background:lawgreen">,
<b style="background:aquamarine">,<b style="background:magenta">,
<b style="background:palegreen">,<b style="background:coral">,
<b style="background:wheat">,<b style="background:khaki">,
<b style="background:lime">,<b style="background:deepskyblue">]]></str>
<str name="hl.tag.post"><![CDATA[</b>]]></str>
</lst>
</fragmentsBuilder>

<boundaryScanner name="default"
        default="true"
        class="solr.highlight.SimpleBoundaryScanner">
<lst name="defaults">
<str name="hl.bs.maxScan">10</str>
<str name="hl.bs.chars">.,!? &#9;&#10;&#13;</str>

```

```

</lst>
</boundaryScanner>

<boundaryScanner name="breakIterator"
                  class="solr.highlight.BreakIteratorBoundaryScanner">
  <lst name="defaults">
    <!-- type should be one of CHARACTER, WORD(default), LINE and SENTENCE -->
    <str name="hl.bs.type">WORD</str>
    <!-- language and country are used when constructing Locale object. -->
    <!-- And the Locale object will be used when getting instance of BreakIterator -->
    <str name="hl.bs.language">en</str>
    <str name="hl.bs.country">US</str>
  </lst>
</boundaryScanner>
</highlighting>
</searchComponent>

<!-- updateRequestProcessorChain(更新请求处理链)
      updateRequestProcessorChain 用于定义多个 updateRequestProcessor 的执行流程,
      严格按照定义顺序依次执行
      http://wiki.apache.org/solr/UpdateRequestProcessor
-->

<!-- Add unknown fields to the schema
这是一个猜测域类型的 update processor, 它会试图将字符串类型的域值
转换成 Booleans, Longs, Doubles, or Dates, 然后添加到 schema.xml 中。
这个示例需要你將 schemaFactory 配置为 ManagedIndexSchemaFactory 并且
指定 mutable=true。
具体请参阅官方 Wiki: http://wiki.apache.org/solr/GuessingFieldTypes
-->
<updateRequestProcessorChain name="files-update-processor">
  <!-- UUIDUpdateProcessorFactory will generate an id if none is present in the incoming
document -->
  <processor class="solr.UUIDUpdateProcessorFactory" />

  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RemoveBlankFieldUpdateProcessorFactory" />
  <processor class="solr.FieldNameMutatingUpdateProcessorFactory">
    <str name="pattern">[^\w-\.]</str>
    <str name="replacement">_</str>
  </processor>
  <processor class="solr.ParseBooleanFieldUpdateProcessorFactory" />
  <processor class="solr.ParseLongFieldUpdateProcessorFactory" />
  <processor class="solr.ParseDoubleFieldUpdateProcessorFactory" />
  <processor class="solr.ParseDateFieldUpdateProcessorFactory">
    <arr name="format">
      <str>yyyy-MM-dd'T'HH:mm:ss.SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss.SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss.SSS</str>
    </arr>
  </processor>

```

```

<str>yyyy-MM-dd'T'HH:mm:ss,SSS</str>
<str>yyyy-MM-dd'T'HH:mm:ssZ</str>
<str>yyyy-MM-dd'T'HH:mm:ss</str>
<str>yyyy-MM-dd'T'HH:mmZ</str>
<str>yyyy-MM-dd'T'HH:mm</str>
<str>yyyy-MM-dd HH:mm:ss.SSSZ</str>
<str>yyyy-MM-dd HH:mm:ss,SSSZ</str>
<str>yyyy-MM-dd HH:mm:ss.SSS</str>
<str>yyyy-MM-dd HH:mm:ss,SSS</str>
<str>yyyy-MM-dd HH:mm:ssZ</str>
<str>yyyy-MM-dd HH:mm:ss</str>
<str>yyyy-MM-dd HH:mmZ</str>
<str>yyyy-MM-dd HH:mm</str>
<str>yyyy-MM-dd</str>
</arr>
</processor>
<processor class="solr.AddSchemaFieldsUpdateProcessorFactory">
  <str name="defaultFieldType">strings</str>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Boolean</str>
    <str name="fieldType">booleans</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.util.Date</str>
    <str name="fieldType">tdates</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Long</str>
    <str name="valueClass">java.lang.Integer</str>
    <str name="fieldType">tlongs</str>
  </lst>
  <lst name="typeMapping">
    <str name="valueClass">java.lang.Number</str>
    <str name="fieldType">tdoubles</str>
  </lst>
</processor>

<processor class="org.apache.solr.update.processor.LangDetectLanguageIdentifierUpdate-
ProcessorFactory">
  <lst name="defaults">
    <str name="langid.fl">content</str>
    <str name="langid.langField">language</str>
  </lst>
</processor>

<processor class="solr.StatelessScriptUpdateProcessorFactory">
  <str name="script">update-script.js</str>
  <!--<lst name="params">-->
  <!--<str name="config_param">example config parameter</str>-->

```

```

<!--</lst>-->
</processor>

<processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>
<!-- Deduplication(索引文档去重检查)
这是一个索引文档去重检查的 update processor 配置示例, 它会基于配置
的其他域的值生成一个 hash 值创建一个 id 域
-->
<!--
<updateRequestProcessorChain name="dedupe">
<processor class="solr.processor.SignatureUpdateProcessorFactory">
<bool name="enabled">true</bool>
<str name="signatureField">id</str>
<bool name="overwriteDups">false</bool>
<str name="fields">name, features, cat</str>
<str name="signatureClass">solr.processor.Lookup3Signature</str>
</processor>
<processor class="solr.LogUpdateProcessorFactory" />
<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
-->

<!-- Language identification(语言识别)
在指定的域上检测当前使用语言, 并创建一个 language_s 的域
关于 langId 的详细信息请参阅
http://wiki.apache.org/solr/LanguageDetection
-->
<!--
<updateRequestProcessorChain name="langid">
<processor class="org.apache.solr.update.processor.TikaLanguageIdentifierUpdat
eProcessorFactory">
<str name="langid.fl">text, title, subject, description</str>
<str name="langid.langField">language_s</str>
<str name="langid.fallback">en</str>
</processor>
<processor class="solr.LogUpdateProcessorFactory" />
<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
-->

<!-- Script update processor(基于 JavaScript 实现的 update processor)
详情请参阅: http://wiki.apache.org/solr/ScriptUpdateProcessor
-->
<!--
<updateRequestProcessorChain name="script">
<processor class="solr.StatelessScriptUpdateProcessorFactory">
<str name="script">update-script.js</str>
<lst name="params">
<str name="config_param">example config parameter</str>

```

```

</lst>
</processor>
<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
-->

```

<!-- Response Writers (响应输出器配置)

这个是用来配置 Solr 响应数据转换类, JSONResponseWriter 就是把 HTTP 响应数据转成 JSON 格式, content-type 即 response 响应头信息中的 content-type, 即告诉客户端返回的数据的 MIME 类型为 text/plain, 且 charset 字符集编码为 UTF-8.

内置的响应数据转换器还有 velocity, xslt 等, 如果你想自定义一个基于 FreeMarker 的转换器, 那你需要实现 Solr 的 QueryResponseWriter 接口, 模仿其他实现类, 你懂的, 然后在 solrconfig.xml 中添加类似的 <queryResponseWriter> 配置即可

<http://wiki.apache.org/solr/QueryResponseWriter>  
Response Writers 通过 wt 请求参数来指定, 如果 wt 参数没有指定, 那么 wt 参数默认值为 default

```

-->
<!-- Solr 默认隐式的配置了如下 ResponseWriter 实例 -->
<!--
<queryResponseWriter name="xml"
                      default="true"
                      class="solr.XMLResponseWriter" />
<queryResponseWriter name="json" class="solr.JSONResponseWriter"/>
<queryResponseWriter name="python" class="solr.PythonResponseWriter"/>
<queryResponseWriter name="ruby" class="solr.RubyResponseWriter"/>
<queryResponseWriter name="php" class="solr.PHPResponseWriter"/>
<queryResponseWriter name="phps" class="solr.PHPSerializedResponseWriter"/>
<queryResponseWriter name="csv" class="solr.CSVResponseWriter"/>
<queryResponseWriter name="schema.xml" class="solr.SchemaXmlResponseWriter"/>
-->
<!-- JSONResponseWriter 配置 -->
<queryResponseWriter name="json" class="solr.JSONResponseWriter">
<str name="content-type">application/json; charset=UTF-8</str>
</queryResponseWriter>

<!-- 自定义的 Response Writer 配置 -->
<queryResponseWriter name="velocity" class="solr.VelocityResponseWriter" startup=
"lazy">
<str name="template.base.dir">${velocity.template.base.dir:}</str>
</queryResponseWriter>

<!-- XSLTResponseWriter 配置 -->
<!--
XSLTResponseWriter 支持根据在 core/conf 目录下找到的 xslt 文件来转换 xml 并输出。
xsltCacheLifetimeSeconds 参数表示每间隔指定的秒数去检查 xslt 文件是否有更新
-->
<queryResponseWriter name="xslt" class="solr.XSLTResponseWriter">
<int name="xsltCacheLifetimeSeconds">5</int>

```

```

</queryResponseWriter>
<!-- Query Parsers (Query 解析器配置)
详情请参阅: https://cwiki.apache.org/confluence/display/solr/Query+Syntax+and+Parsing
多个 QParserPlugins 可以通过 name 来注册, 然后在 QueryComponent 组件里通过 defType
参数或者 LocalParams 参数来使用该插件
-->
<!-- 自定义 queryParser 配置示例 -->
<!--
<queryParser name="myparser" class="com.mycompany.MyQParserPlugin"/>
-->
<!-- Function Parsers 功能查询解析器配置
详情请参阅: http://wiki.apache.org/solr/FunctionQuery
多个 ValueSourceParsers 可以通过 name 来注册, 然后当使用 "func" QParser
可以通过使用 Function name 来使用它
-->
<!-- 自定义的 function parser 配置示例 -->
<!--
<valueSourceParser name="myfunc" class="com.mycompany.MyValueSourceParser" />
-->
<!-- Document Transformers (文档转换器配置)
详情请参阅:
http://wiki.apache.org/solr/DocTransformers
-->
<!--
<transformer name="db" class="com.mycompany.LoadFromDatabaseTransformer" >
<int name="connection">jdbc:// ....</int>
</transformer>
为每个 document 添加一个常量
<transformer name="mytrans2" class="org.apache.solr.response.transform.Value-
AugmenterFactory" >
<int name="value">5</int>
</transformer>
为每个 document 添加一个默认值
<transformer name="mytrans3" class="org.apache.solr.response.transform.Value-
AugmenterFactory" >
<double name="defaultValue">5</double>
</transformer>
如果你正在使用 QueryElevationComponent, 你可能希望标记某个文档的权重要更高点那么你需要配置
EditorialMarkerFactory 如下所示:
<transformer name="qecBooster" class="org.apache.solr.response.transform.Editorial-
MarkerFactory" />
-->
<!--
Solr 管理以及 Solr Web UI 相关的配置
getTableFiles 定义了哪些文件可以通过 web 接口被访问
pingQuery 配置了一个 ping 查询用于监听 Solr Server 的健康状态。ping 查询的请求 URL 为 /
admin/ping。
他可以用于集群中的 Solr server 节点通过检测所有节点的响应时间来进行负载均衡。
-->
<admin>

```



```

<defaultQuery>solr</defaultQuery>
<gettableFiles>
    solrconfig.xml
    schema.xml
</gettableFiles>
<pingQuery>q=solr&version=2.0&start=0&rows=0</pingQuery>
<!-- 为负载均衡器配置一个心跳检测文件，通过 ping query 去检测 healthcheck file 是否存在，
若不存在则认为这个节点挂掉了，负载均衡器根据 ping query 返回结果去决定是否需要将该节点从集群
中移除
-->
<healthcheck type="file">server-enabled</healthcheck>
</admin>
</config>

```

最后需要说明下的是在 solrconfig.xml 中有大量类似 `<arr><list><str><int>` 这样的自定义标签，下面做个统一的说明，如图 3-1 所示。

| Element                    | Description                             | Example   |
|----------------------------|---|---|
| <code>&lt;arr&gt;</code>   | Named, ordered array of objects         | <pre> &lt;arr name="last-components"&gt;   &lt;str&gt;spellcheck&lt;/str&gt; &lt;/arr&gt; </pre>  |
| <code>&lt;lst&gt;</code>   | Named, ordered list of name/value pairs | <pre> &lt;lst name="defaults"&gt;   &lt;str name="omitHeader"&gt;true&lt;/str&gt;   &lt;str name="wt"&gt;json&lt;/str&gt; &lt;/lst&gt; </pre> |
| <code>&lt;bool&gt;</code>  | Boolean value—true or false             | <code>&lt;bool&gt;true&lt;/bool&gt;</code>  |
| <code>&lt;str&gt;</code>   | String value                            | <pre> &lt;str&gt;spellcheck&lt;/str&gt; or &lt;str name="wt"&gt;json&lt;/str&gt; </pre>   |
| <code>&lt;int&gt;</code>   | Integer value                           | <code>&lt;int&gt;512&lt;/int&gt;</code>   |
| <code>&lt;long&gt;</code>  | Long value                              | <code>&lt;long&gt;1359936000000&lt;/long&gt;</code>   |
| <code>&lt;float&gt;</code> | Float value                             | <code>&lt;float&gt;3.14&lt;/float&gt;</code>  |

图 3-1 solrconfig.xml 中的自定义标签

这张图摘自于《Solr in action》这本书，由于是英文的，所以我稍微解释下：

- arr: array 的缩写，表示一个数组，name 即表示这个数组参数的变量名；
- lst: list 的缩写，但注意它里面存放的是 key-value 键值对；
- bool: 表示一个 boolean 类型的变量，name 表示 boolean 变量名，同理还有 int, long, float, str 等；
- Str: string 的缩写，唯一要注意的是 arr 下的 str 子元素是没有 name 属性的，而 list 下的 str 元素是有 name 属性的。

## 3.3 schema.xml 配置详解

### 3.3.1 Solr Schema 设计思想

用户输入的数据传递给 Solr, Solr 会按照一定的模式去创建索引文档, 而模式规定了每个索引文档包含的域, 以及域的类型、采用什么分词器分词等, 这个模式就称为 Schema。Scheme 决定了索引是如何创建并存储, 同时 Solr 里的 Schema 是一种相对松散的结构, 对比来看, 关系型数据库里的 Table 其实也有可以的一套 Schema, 但数据库的 Schema 规定表的每一条 record 拥有的列必须保持一致, 这点 Solr 相对宽松些, 因为 Solr 里可以配置每个域不是必需的, 非必需域可以缺失且对索引创建和查询没有任何影响。Solr 根据 Schema 构建索引的过程跟我们小时候使用的《新华字典》编著过程是一样的。我们想象一下, 假如自己编著一本中文字典, 我们应该怎么做? 首先应该按照字母表将每个汉字的正确书写、读音、拼音、词性、释义、例句、同义词等一页页印刷, 并给每一页编上页码。但为了方便以后快速找到某个字的详细释义, 我们需要将每个字对应的页码做成一个目录, 以后要查阅某个字直接先查阅目录找到该字的所在页码, 最后直接翻到那一页即可。当然现实中, 字典目录还需要按照偏旁部首、拼音、笔画对目录进行分类。制作字典目录的过程其实就好比 Solr 的索引创建过程, 根据目录查汉字就好比 Solr 的 Query (查询), 而每个字需要印刷的正确书写、读音、拼音、词性、释义、例句、同义词这部分信息其实就是模式, 每个汉字几乎都是按照这个模式来编排内容的, 定义字典里每个汉字需要展示内容的模板跟 Solr 的 Schema 设计思想如出一辙。

Solr 允许你创建一个包含多个域的索引, 上面列举的示例展示了 Solr 是如何创建索引的, 而 Schema 就是提供用户来定义内容编排模板从而干预 Solr 创建索引过程。

### 3.3.2 Solr 眼里的世界

Solr 里信息的基本单元是 Document, Document 表示描述事物的一组数据集合。就好比新华字典里每个汉字应该包含之前列举的信息。再比如一个关于 People 的 Document 或许应该包含姓名、个人简介、爱好颜色、鞋尺码等信息, 一个关于 Book 的 Document 或许应该包含书名、作者、出版年份、总页数等信息。

在 Solr 的世界里, Document (文档) 由多个 Field (域) 组成, 而域携带了部分信息, 比如鞋子尺码应该是一个域, 姓名也应该是一个域。Field (域) 可以包含不同类型的数据, 比如姓名是一个文本域, 而出版年份是一个数字域, 比如 2008 表示该书于 2008 年出版。显然, 域的定义是很灵活的, 出版年份既可以定义为文本域也可以定义为数字域。但为你的域定义正确的类型, 有助于 Solr 更好的解析你的数据并创建索引, 而后你的用户也能通过查询获取到更精准的反馈结果。

你可以通过指定 Field (域) 的 FieldType (域类型) 来告诉 Solr 你数据的类型。FieldType 指示了 Solr 应该如何去解析 Field 以及 Field 是如何被查询的。

当你添加一个 Document, Solr 从 Document 的 Field 里提取出信息,然后将这些信息添加到索引中。当你执行一个 Query (查询), Solr 可以快速查阅索引然后返回匹配到的文档。

### 3.3.3 域分词

当创建索引时,域分词告诉 Solr 应该如何处理输入的数据。这个过程更准确的说法是 processing 或 even digestion,但官方称为 analysis。

考虑一下,比如在 person 的 document 里的个人简历这个域,它的每个单词必须被索引,这样你就可以快速地找到那个人。但个人简历里你并不是每个单词都会关心,比如“the”“a”“to”,而且你也不想因此增大索引的体积。此外,假如个人简历域包含一个单词“Cute”,那么假定用户输入的搜索关键词是“cute”,然而个人简历域包含“Cute”的 people 并没有被返回,这显然不是我们想要的结果。

解决上面提出的问题的方案就是 field analysis(域分词)。对于上面例子中的“个人简历”域,你可以告诉 Solr 将其分解成一个个单词,然后你可以告诉 Solr 将每个单词转换成小写形式并移除标点符号。

Field analysis (域分词)是域类型的重要部分,域分词一般发生域数据摄入阶段,比如在索引创建时、查询时。Analyzer (分词器)会检索域的文本并生成一个 TokenStream, Analyzer (分词器)应该是一个单例,它由 1 个 Tokenizer + N 个 TokenFilter 组成。

tokenizer 的职责是就是打散文本输入流生成 token 列表,每个 token 就是原始文本的一个字符子序列, analyzer (分词器)关心的是自己配置为哪个 Field,而 tokenizer 关心的是 Reader 输入流, Tokenizers 从 Reader 实例读取数据然后生成一个 TokenStream。TokenStream 里包含的每个 Token 除了携带了字符文本内容,还额外包含了 token 在域中的出现位置,当你使用高亮组件时会需要它。

TokenFilter 接收一个 TokenStream 并缓存它,然后经过一系列的转换、剔除操作,最终会返回一个新的 TokenStream,一个 TokenFilter 的输出将会是下一个 TokenFilter 的输入,从而可以构成一个 pipeline (管道)或 chain (链条)。比如 tokenizer 加一系列 TokenFilter 就构成了 Analyzer (分词器)。类似 tokenizer, TokenFilter 接收一个 TokenStream 输入流并生成一个新的 TokenStream。不同于 tokenizer 的是, TokenFilter 的输入是另一个 TokenStream,而 tokenizer 的输入是一个 Reader。

### 3.3.4 Solr 的 schema 文件


Solr 在 Schema 文件里存储域类型 (FieldType) 和域 (Field) 的详细信息, Schema 文件的名称以及存储位置取决于你是如何初始化配置你的 Solr。

如果你的 Schema 文件名称为 managed-schema,那么 Solr 默认会支持通过 Schema API 对此 Schema 文件的运行时更新,要想实现 Schema 文件运行更新还有两种可选的方式: Schemaless Mode (无 Schema 模式),或者显式的配置 ManagedIndexSchemaFactory (在 solrconfig.xml

中配置)。如果你的 Schema 文件名称为 Schema，那么 Schema 文件必须手动修改并重新加载才能应用该方法。在 SolrCloud 模式下，你只能通过 Schema API 来访问 Schema 文件。

不论你的 Schema 文件名称是什么，它们的文件结构都是一样的，只是你与它的交互方式改变了。如果使用 managed-schema，那么意味着你只能通过 Schema API 来访问 Schema 文件且无法人工编辑 Schema 文件。如果使用的不是 managed-schema，那么你能人工编辑 Schema 文件，即便你开启 Schema API，也无法通过 Schema API 来访问 Schema 文件。

---

 **注意** 如果你正在使用 SolrCloud 但并没有使用 Schema API，那么需要通过 ZooKeeper 的 upconfig（上传配置）和 downconfig（下载配置）命令来获取 Schema 文件。

---

### 3.3.5 Solr 的域类型

域类型定义了 Solr 应该如何从域中解析数据以及域是怎样被查询的。在 Solr 内部默认定义了很多域类型，如 StringField、TextField、BoolField。

一个域类型定义包含 4 种类型信息：

- ❑ name：域类型的名称，强制性必须指定的，且必须保证同一个 schema 文件内唯一性；
- ❑ class：域类型对应的 Java 类，强制性必须指定的，如果是 Solr 内置的域类型，使用 solr. 类名方式指定即可，如果是自定义的域类型，你必须指定类的完整包路径；
- ❑ 如果你配置的域类型为 TextField，那么你还需要配置 <analyzer> 元素来指定分词器；
- ❑ Field type properties：域类型的属性依赖于具体实现类，某些属性是强制性的。

域类型在 schema.xml 文件中定义，每个域类型都是通过 <fieldType> 元素来定义的，他们还可以选择性的定义在 <types> 元素内，从而能够对于 FieldType 进行分组管理。下面是一个名称为 text\_general 域类型的定义示例，代码如下所示：

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.
txt" />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.
txt" />
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignore-
Case="true" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

上面示例的第一行中我们定义一个域类型名称为 `text_general`，域类型的实现类为 `solr.TextField`。剩下部分为域分词相关配置。域类型的实现类必须确保配置正确了，在 `schema.xml` 中，类名的 Solr 前缀其实是 `org.apache.solr.schema` 或 `org.apache.solr.analysis` 的一个缩写，当前的前提是该域类型的实现类是 Solr 内置的，如果是自定义的域类型，那你必须指定域类型实现类的完整包路径哦。


域类型的实现类决定了大部分域类型的行为，但仍有一些可选的属性可以被定义，比如下面的 `date field` 定义了两个属性：`sortMissingLast`、`omitNorms`。

```
<fieldType name="date" class="solr.TrieDateField"sortMissingLast="true" omitNorms="true"/>
```

域类型支持的通用属性如表 3-1 所示。

表 3-1 Solr 域类型支持的通用属性

属性	描 述
name	域类型的名称
class	域类型的实现类，内置的域类型实现类以 <code>solr.</code> 为前缀
positionIncrementGap	配置多值之间额间隙，用于阻止伪短语匹配，此配置适用于多值域
autoGeneratePhraseQueries	若设置为 <code>true</code> ，Solr 会自动为相邻的 <code>term</code> 生成短语查询，若设置 <code>false</code> ， <code>term</code> 必须被一对双引号包裹才能被认为是一个短语查询。此配置是针对于 <code>TextField</code> 的
docValuesFormat	配置 <code>DocValuesFormat</code> 实现。这个配置需要你提前在 <code>solrconfig.xml</code> 中配置 <code>SchemaCodecFactory</code>
postingsFormat	<code>postingsFormat</code> 表示用于编/解码 <code>Term</code> 、 <code>Postings</code> 、数据的实现类，属于 <code>Lucene</code> 比较底层的 API，此参数可选值请查阅 <code>PostingsFormat</code> 类的实现子类，一般不需要显式指定此参数，保持默认即可。除非你想要自定义 <code>PostingsFormat</code> 实现 这个配置需要你提前在 <code>solrconfig.xml</code> 中配置 <code>SchemaCodecFactory</code>

 **注意** `Lucene` 索引的向旧版本兼容的特性只支持默认的 `codec`，如果你选择了在你的 `schema.xml` 中自定义 `postingsFormat` 或 `docValuesFormat`，升级 Solr 版本可能要求你在升级之前，切换到默认的 `codec` 实现并优化你的索引或者在升级 Solr 版本之后，直接重建所有索引。

域类型下还可以指定一个 `<similarity>` 元素，`<similarity>` 表示配置一个打分器，它会影响相关域的文档评分，任何未定义 `<similarity>` 的域类型默认都是使用 `DefaultSimilarity` (`Solr6` 中默认 `Similarity` 实现已变更为 `BM25Similarity`)。

Solr 内置提供了如下预定义域类型，它们都在 `org.apache.solr.schema` 包下，具体如表 3-2 所示。

表 3-2 Solr 内置提供的预定义域类型

类名	描 述
BinaryField	表示经过 base64 编码的字符串域类型，即你需要把 binary 数据进行 base64 编码才能被 Solr 进行索引
BoolField	用于表示 boolean 类型的数据，除了 true、false，第一个字符是 "l", "t", or "T" 也会被认为是 true
CollationField	支持 Unicode 排序规则进行排序和区间范围查询，当你使用 ICU4J 时，选择使用 ICUCollationField 是更好的选择
CurrencyField	支持货币以及货币兑换和货币汇率换算
DateRangeField	支持对日期范围进行索引，从而支持日期范围查询
ExternalFileField	对外部文件进行索引的域类型，它支持从外部硬盘上读取指定文件的内容创建索引
EnumField	对枚举进行索引的域类型
ICUCollationField	支持 Unicode 排序规则进行排序和区间范围查询
LatLonType	用于对经纬度数据进行索引，经纬度数据格式必须为 latitude,longitude，纬度在前，经度在后，两者之间使用逗号分割，且两者之间不能有空格等其他字符 这个域用于 Solr 中的 Spatial Search（空间查询），注意：LatLonType 不支持多值域
PointType	支持对 N 个维度的坐标点数据的索引，比如数学里的二维坐标 x,y，甚至立体几何的三维坐标 x,y,z 注意：PointType 不支持多值域
PreAnalyzedField	即预分词域，它提供一个序列化的 token stream 传递给 Solr，这样 Solr 在对 PreAnalyzed-Field 创建索引时不再需要进行分词了，因为如何分词处理已经提前在序列化的 token stream 中定义好了
RandomSortField	这个域类型不包含域值，它用于查询时，根据 RandomSortField 排序时，将会对返回结果集进行随机排序，一般与 dynamicField 搭配使用
SpatialRecursivePrefixTreeFieldType	用于深度遍历前缀树的 FieldType，主要用于获取 Lucene 中的 RecursivePrefixTreeStrategy。这个域用于 Solr 中的 Spatial Search（空间查询）
StrField	对 UTF-8 编码的字符串或者 Unicode 字符进行索引，StrField 不会分词，且 StrField 有个硬性限制，字符大小必须小于 32KB。它支持 docValues 域，但当为其添加了 docValues 域，则要求只能是单值域且该域必须存在或者该域有默认值
TextField	TextField 用于对长文本进行索引，与 StrField 最大区别是，TextField 会分词
TrieDateField	Date 域类型，表示一个毫秒精度的时间点，启用 precisionStep="0" 会提升 Date 域的排序性能并最大限度地减少索引大小 配置 precisionStep="8" 会提升 Date 域的范围区间查询性能
DateRangeField	用于对日期范围进行索引，并且基本兼容 TrieDateField
TrieDoubleField	Double 域类型 precisionStep="0" 会提升 Double 域的排序性能并最大限度地减少索引大小； 配置 precisionStep="8" 会提升 Double 域的范围区间查询性能
TrieField	如果你使用这个域，那么还需要指定一个 type 属性，它的可选值为 integer、long、float、double、date。使用这个域跟使用 TrieXXXField 是一样的
TrieFloatField	Float 域类型，同 TrieDoubleField 类似
TrieIntField	Integer 域类型，同 TrieDoubleField 类似
TrieLongField	Long 域类型，同 TrieDoubleField 类似



(续)

类名	描 述
UUIDField	UUID 域，如果域值为空字符串或 Null 或字符串“NEW”，那么 Solr 会自动生成 UUID 字符串并创建索引，如果用户设置了域值，那么 Solr 会首先检查域值是否符合 UUID 规范，如果不符合 UUID 规范，那么会抛出异常，否则会将域值转换为小写形式，最后创建索引。 在 SolrCloud 环境下，配置一个 UUIDField 并且指定域值为“NEW”是不建议的，此时建议使用 UUIDUpdateProcessorFactory 来代替

1. CurrencyField 的使用

CurrencyField（货币域）提供了对货币值的支持，Solr 可以在查询时使用它完成货币的兑换和货币汇率换算。CurrencyField（货币域）支持以下功能：

- ❑ 区间范围查询；
- ❑ Function 区间范围查询；
- ❑ 排序；
- ❑ 货币代号以及货币符号（比如美元的 \$，人民币的 ¥）的解析；
- ❑ 支持对称 & 非对称汇率。

CurrencyField 配置示例如下：

```
<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
  defaultCurrency="USD" currencyConfig="currency.xml" />
```

在上面示例中，我们定义了域类型的名称和 class 类，并定义当前默认货币 defaultCurrency=USD（即美国的美元），还使用 currencyConfig 属性定义了货币配置文件 currency.xml 的加载路径。currency.xml 文件里定义了当前默认货币与其他货币之间的汇率，还有另一种可选的实现就是使用 OpenExchangeRatesOrgProvider 提供的远程接口来获取货币数据。

在创建索引时，可以使用本地货币来索引货币域，举个例子，假如一个电商网站上的商品价格显示的是欧元，你可以这样索引它：“1000, EUR”或者“100”，如果域值包含了逗号，Solr 会通过逗号对域值进行分割，第一个值就被当作货币的数值，第二个值被当作是货币的类型（比如加拿大的加元是 CAD）。如果域值你仅仅只提供了一个数字，那么货币类型就以 defaultCurrency 为准，如果 <FieldType> 里 defaultCurrency 属性也没定义，那么默认货币就以 CurrencyField 类源码里的 DEFAULT\_DEFAULT\_CURRENCY=“USD”常量为准。

currency.xml 配置示例如下：

```
<currencyConfig version="1.0">
  <rates>
    <!-- Updated from http://www.exchangerate.com/ at 2011-09-27 -->
    <rate from="USD" to="ARS" rate="4.333871" comment="ARGENTINA Peso" />
    <rate from="USD" to="AUD" rate="1.025768" comment="AUSTRALIA Dollar" />
    <rate from="USD" to="EUR" rate="0.743676" comment="European Euro" />
```

```

<rate from="USD" to="BRL" rate="1.881093" comment="BRAZIL Real" />
<rate from="USD" to="CAD" rate="1.030815" comment="CANADA Dollar" />
<rate from="USD" to="CLP" rate="519.0996" comment="CHILE Peso" />
<rate from="USD" to="CNY" rate="6.387310" comment="CHINA Yuan" />
<rate from="USD" to="CZK" rate="18.47134" comment="CZECH REP. Koruna" />
...//and so on
</rates>
</currencyConfig>

```

配置货币域的货币汇率你可以通过指定一个 provider 来完成，Solr 默认内置了两种 provider 实现：FileExchangeRateProvider 和 OpenExchangeRatesOrgProvider。其中：

FileExchangeRateProvider 是默认值，可以不用显式指定。OpenExchangeRatesOrgProvider 表示从一个远程的接口获取货币汇率数据，你可以通过给 FieldType 添加一个 providerClass 属性来指定 provider，具体配置示例如下所示：

```

<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
providerClass="solr.OpenExchangeRatesOrgProvider"
refreshInterval="60"
ratesFileLocation="http://www.openexchangerates.org/api/latest.json?app_id=your-
PersonalAppIdKey"/>

```

- providerClass：表示 provider 实现类，内置的 provider 只需以 solr. 开头来代替默认包名即可，如果是自定义实现的 provider 类，你则需要指定完整的包路径；
- refreshInterval：表示每间隔多少分钟再重新从远程接口获取一次汇率数据；
- ratesFileLocation：获取货币汇率数据的远程接口 URL。

## 2. TrieDateField 的使用

TrieDateField 用于对日期时间类型数据进行索引，日期时间类型数据在 Java 里可以使用 java.util.Date、Long、Integer 来表示，Long、Integer 类型可以表示毫秒数，当然字符串类型也可以，但字符串类型来表示日期时间在 Solr 里用 StrField 即可。所以只要你日期时间数据是表示毫秒数的数字（比如 Long）或者 java.util.Date，那么就可以使用 TrieDateField。

TrieDateField 可以用于表示毫秒精度的时间点，且只能精确到毫秒精度。Solr 对于日期时间格式有严格要求，在处理日期时间字符串与 java.util.Date 之间转换时，Solr 使用的日期时间字符串格式严格遵循 XML Schema 规范中对于 dateTime 类型的权威性约束形式，格式如下：

```
YYYY-MM-DDThh:mm:ssZ
```

- YYYY 表示年份；
- MM 表示月份，两位数字表示；
- DD 表示日期，两位数字表示；
- Hh 表示小时数，24 小时制；
- mm 表示分钟数；

- ss 表示秒数;
- Z 表示 UTC 时区。



**注意** Solr 里 `TrieDateField` 的时区无法设置覆盖, 默认时区是 UTC, 且无法修改。由于我们目前处在东八区即 Solr 里的 UTC 时间再加 8 小时才等于我们的北京时间。

Solr 里对于日期时间存储只能精确到毫秒, 任何超出毫秒精度部分的数据将会被丢弃, 如果你需要精确到毫秒, 那么你的日期时间数据格式应该类似这样:

```
1972-05-20T17:33:18.772Z
```

```
1972-05-20T17:33:18.77Z
```

```
1972-05-20T17:33:18.7Z
```

对于这种日期时间格式, 新手可能会感觉到不适应, 毕竟这种格式不是国人日常习惯的风格, 为此, 可能有人会有这样的疑问: 2016-08-01T08:23:45.678Z 这样的日期时间字符串如何转换成 UTC 时区的 `Date` 对象呢?

```
String input = "2016-08-01T08:23:45.678Z";
TimeZone utc = TimeZone.getTimeZone("UTC");
SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
f.setTimeZone(utc);
GregorianCalendar cal = new GregorianCalendar(utc);
cal.setTime(f.parse(input));
Date date = cal.getTime();
```

通过上面的代码, 我们就能够方便地将 Solr 规范的日期时间格式字符串转换成 `java.util.Date` 类型, 而 `java.util.Date` 类型正是 Solr 的 `TrieDateField` 能够接收的数据类型。由于 `TrieDateField` 还能够接收数字类型的毫秒数, 此时你可以再将 `java.util.Date` 转换成 `Long` 类型。

一个 `TrieDateField` 类型完整配置示例如下所示:

```
<field name="Date" class="solr.TrieDateField" datatypeformat="yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
indexed="true" multivalued="false" stored="true" precisionStep="6"
type="date">

</field>
```

- name: 表示域类型的名称;
- class: 表示域类型的实现类;
- datatypeformat: 表示 `TrieDateField` 域的域值最终显示的字符串格式;
- indexed: 表示是否对当前 `TrieDateField` 域创建索引;
- multivalued: 表示是否为多值域;
- stored: 表示是否存储域值;

□ type: 表示 TrieDateField 域的原始域值的数据类型, 可选值有

- INTEGER;
- LONG;
- FLOAT;
- DOUBLE;
- DATE。

type 参数值不区分大小写, 因为你不管是大写还是小写, 最终 Solr 内部都会先将其统一转换成大写形式, 然后跟 INTEGER、LONG、FLOAT、DOUBLE、DATE 进行比较来确定 TrieDateField 域的域值原始数据类型, 如果用户提供的 type 参数不属于规定的取值范围内, 那么会抛异常。如果用户没有指定 type 参数, 那么默认 Solr 源码内部会按 type=date 来处理, 即默认 TrieDateField 域会将域值当做 java.util.Date 类型来处理。

不管是 java.util.Date 还是表示毫秒数的 Long/Integer/Float/Double, TrieDateField 域最终域值都会转换成数值来处理, 其中 java.util.Date 类型的域值会被转换成 Long 类型的毫秒数, 所以 TrieDateField 本质就是一个数字域, 具体请参阅 Solr 的 TrieField 类的 createField 方法部分源码。了解了这点, 你就明白了为什么 TrieDateField 还有个 precisionStep 配置参数, 因为 TrieDateField 的域值最终都是转换成数字, 为了提升数字区间查询性能, 需要使用 precisionStep 参数指定的精度步长来对原始域值进行分割额外生成 N 个 Term, 再对生成的这些 Term 创建索引假如 TrieDateField 的域值数据类型是 Integer, 而一个 Integer 占 4 个字节共 32bit, 若 precisionStep=4, 那么会生成这些 Term: 前 4 个 bit、前 8 个 bit、前 12 个 bit、前 16 个 bit、前 20 个 bit、前 24 个 bit、前 28 个 bit、全部 32 个 bit (即原始域值)。由此, 我们可以知道, precisionStep 的值越大, 生成的 Term 个数越少, 索引体积也就越小, 但区间范围查询性能就越低, 反之, precisionStep 的值越小, 生成的 Term 个数越多, 索引体积也就越大, 但区间范围查询性能就越高。至于为什么增大 precisionStep 值能提升区间查询性能, 之前的章节有提到过, 这里就略过了。

### 3. DateRangeField 的使用

DateRangeField 几乎是 TrieDateField 的一个替代品, DateRangeField 基本兼容 TrieDateField, 两者唯一区别是 XML 格式的源数据或者 SolrJ 的响应格式会以 String 类型暴露域值而不是 Date 类型。DateRangeField 域底层的数据可能会有点大, 但 DateRangeField 第二次查询的速度要比 TrieDateField 快, 但 DateRangeField 的最重要关键点是它允许对日期范围数据进行索引, 正如它的名称一样, 同时 DateRangeField 支持指定索引数据与查询范围之间的 3 种关系: Intersects (默认值), Contains, Within, 你可以在查询时通过 op 参数来指定。

使用 DateRangeField 你可以直接对日期范围字符串进行索引, 日期范围字符串必须是类似这样的: [2000 TO 2014-05-21T17:33:18.772Z], 并且支持日期范围的多值域索引。假如没有 DateRangeField, 要实现日期范围查询, 你可能需要定义两个域 date\_start 和 date\_end, 这个时候日期区间查询就显得很简单, 你只需要像这样查询即可:

```
q=date_s:[target TO *] AND date_e:[* TO target]
```

然而，用户查询的范围区间是不固定的，我们是不是得穷举出所有用户可能执行的查询区间并存储到 `date_start` 和 `date_end` 域上？这种做法显然不合理，而 `DateRangeField` 支持日期范围的多值域就是为了解决这类问题。

假定你有这样一个需求：查询出某个时间段内正开放营业的餐厅。需要注意的是，每个餐厅在一天之内可以有多个开放时间。而且每天的开放时间是不固定的。你可以这样做，在 `shcema.xml` 中添加一个 `date_range` 域，示例如下所示：

```
<field name="openingHours" type="date_range" indexed="true" stored="true" multiValued="true" />
<fieldType name="date_range" class="solr.DateRangeField"/>
```

你的 `date_range` 域的数据可能是这样的：

```
[
  "[2016-02-01T03:00Z TO 2016-02-01T15:00Z]",
  "[2016-02-02T03:00Z TO 2016-02-02T15:00Z]",
  "[2016-02-03T03:00Z TO 2016-02-03T15:00Z]",
  "[2016-02-04T03:00Z TO 2016-02-04T15:00Z]",
  "[2016-02-07T03:00Z TO 2016-02-07T15:00Z]",
  "[2016-02-05T03:00Z TO 2016-02-05T16:00Z]",
  "[2016-02-06T03:00Z TO 2016-02-06T16:00Z]"
]
```

需要注意的是，你的日期格式必须要转换成 UTC 时间格式，OK，现在你可以这样查询：

```
facet.range=openingHours&f.openingHours.facet.range.start=NOW&f.openingHours.
facet.range.end=NOW%2B6HOUR&f.openingHours.facet.range.gap=%2B1HOUR
```

- `facet.range` 表示你需要对哪个域进行区间查询；
- `facet.range.start` 表示查询区间的起始值；
- `facet.range.start` 表示查询区间的结束值；
- `facet.range.gap` 表示区间递增的间隙。

#### 4. EnumField 的使用

`EnumField` 用于对枚举类型的数据进行索引，那么何为枚举？下面是 Java 里的一个枚举示例：

```
public enum LogLevel {
    INFO,           // 打印详细信息
    DEBUG,          // 打印 DEBUG 调试信息
    WARNING,        // 打印警告信息
    ERROR           // 打印错误信息
}
```

下面是 `EnumField` 的配置示例，代码如下所示：

```
<fieldType name="logLevelType" class="solr.EnumField" enumsConfig="enums-
Config.xml" enumName="logLevel"/>
```

- ❑ **enumsConfig**: 用于指定枚举配置文件，在其中定义支持的所有枚举值；
- ❑ **enumName**: 表示枚举名称，它必须与枚举配置文件中配置的某一个 `<enum>` 元素的 `name` 属性值保持一致。

通过 `enumsConfig` 参数配置的枚举配置文件里可以包含多个枚举值，它以 `<enumsConfig>` 作为根元素，`<enumsConfig>` 可以包含多个 `<enum>` 元素，每个 `<enum>` 元素表示一个枚举，其下又可以包含多个 `<value>` 元素，每个 `<value>` 元素又表示当前枚举下的所有枚举值。

下面是一个枚举配置文件 `enumsconfig.xml` 示例，代码如下所示：

```
<?xml version="1.0" ?>
<enumsConfig>
  <enum name="logLevel">
    <value>INFO</value>
    <value>DEBUG</value>
    <value>WARNING</value>
    <value>ERROR</value>
  </enum>
  <enum name="priority">
    <value>Not Available</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High</value>
    <value>Urgent</value>
  </enum>
  <enum name="risk">
    <value>Unknown</value>
    <value>Very Low</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High</value>
    <value>Critical</value>
  </enum>
</enumsConfig>
```



**注意** 你不能调整枚举值的顺序、删除已经存在的枚举值，这些操作都是无效的，除非你对枚举域重建索引。但你可以在末尾追加一个新的枚举值。

`EnumField` 首先会加载枚举配置文件，然后解析 XML 里的 `enum` 元素，然后将枚举值和枚举编号存入 `Map` 中，其中枚举值为 `key`，编号为 `value`，编号根据 `<value>` 元素从上至下的定义顺序从零开始计算，同时这个编号值也是实际写入索引的真实值，源码内部实际会创建一个 `IntField`。

`EnumField` 强依赖于枚举配置文件，不能对 `java` 里的 `Enum` 类来创建域，枚举数据只能



通过一个枚举配置文件定义，这点让我很意外！

## 5. ExternalFileField 的使用

ExternalFileField 支持为某个域指定域值，而该域值是从一个外部文件加载得到的。也就是说某个域的域值不需要存储到 Solr 中，可以从外部文件获取。ExternalFileField 是不支持查询的，它只能用于 Function Query 或者显示。

当你想要更新多个文档中的某一个特定的域而不是更新剩下的所有文档时，你可能需要 ExternalFileField。举个例子，假设你的 Document 有一个 views 域，而这个 views 域可能表示的是一篇博客的浏览量，由于浏览量可能每分每秒都在改变，你可能希望每间隔 1 分钟或者半小时单独更新下 views 表示浏览量的域，然而文档的其他域的值可能更新并不是那么频繁，如果没有 ExternalFileField，你可能需要更新每个 Document，然而你仅仅只是更新了每个文档的 views 域。如果你使用了 ExternalFileField，就显得更高效了，因为所有文档的 views 域的域值是存储在外部的一个文件，而且该文件可以按照你的意愿定期更新。

在 schema.xml 中，ExternalFileField 的配置示例如下所示：

```
<fieldType name="viewsFile" keyField="id" defVal="0" stored="false" indexed="false" class="solr.ExternalFileField" valType="pfloat"/>
```

- ❑ keyField：定义唯一性标识域的名称，它通常是 Document 的唯一主键域，但并不是必须的，只要 keyField 能唯一标识一个 Document 即可；
- ❑ defVal：表示当某个 Document 在外部文件中不存在时，ExternalFileField 的域值采用 defVal 定义的默认值；
- ❑ valType 定义了需要在外部文件里加载的域值的实际数据类型，可选值为 pfloat、float、tfloat，这个属性是可选的，可以不用配置；
- ❑ ExternalFileField 依赖的外部文件默认需要存放在 \$SOLR\_HOME/data 目录下，文件名称应该满足如下规则：

external\_fieldname 或者 external\_fieldname.\*

所以，external\_entryRankFile 或者 external\_entryRankFile.txt 是合法的文件名称。如果 external\_fieldname.\* 规则匹配到多个文件，此时会按照文件名称进行排序，会按照排在最后的那一个文件为准。这个外部文件的内容格式如下所示：

```
1=1000
2=20000
3=999
```

左边表示 keyField 参数表示的域的域值，右边表示需要存储到外部文件里而且经常需要更新的域的域值。我们只需要在 field 定义里通过 type="viewsFile" 来应用我们的 ExternalFileField，自此以后更新 views 域的域值再也不需要更新索引的每个 Document，需要的仅仅是周期性的更新这个外部文件里每个键值对里右边的值即可完成 Document 某个

域的原子更新。

## 6. UUIDField 的使用

根据域类型的名称就应该知道，UUIDField 用于对 UUID 字符串进行索引。UUIDField 跟 StrField 很相似，不同的是，UUIDField 会检查用户提供的字符串是不是合法的 UUID 格式，如果用户没有为 UUIDField 指定域值或者指定为空字符串或者指定为固定字符串“NEW”，那么 Solr 会自动为 UUIDField 生成 UUID 字符串。

Solr 的索引并不是必须存在一个唯一主键域，但大部分情况下需要。当你确保你每次添加的索引文档不可能出现重复，即便出现重复，对你的业务来说基本符合而且需求没什么太大影响，那么你就可以不使用唯一主键域。然而当你有以下需求时，那就需要一个唯一主键域：

- ❑ 当你需要增量式的添加索引文档，索引文档可能会重复添加，而你也不希望索引文档出现重复；
- ❑ 当你索引的数据很大，可能不会把它存储到 Solr 索引中，但又希望后续能够根据索引数据中的某个域从外部数据源再次加载该数据；
- ❑ 当你希望更简便的使用唯一主键域来删除一个文档，而不是每次都繁琐的使用 Query 来匹配待删除的文档；
- ❑ 当你使用 DistributedSearch（分布式查询）时，需要一个唯一键来过滤多个分片上返回的重复文档，因为你不希望将重复的文档返回给用户。

在 Solr 中配置生成唯一 UUID，需要修改两个配置文件：

### ❑ schema.xml

```
<fieldType name="uuid" class="solr.UUIDField" indexed="true" />
```

再添加一个 id 域，并应用刚刚定义的 UUIDField 域类型

```
<field name="id" type="uuid" indexed="true" stored="true"
multiValued="false" required="true"/>
```

### ❑ solrconfig.xml

经过上面两个步骤的配置，我们的 UUIDField 域已经配置好了，但 Solr 还提供了一个自动更新 UUIDField 的处理器，也就是说使用它 UUIDUpdateProcessorFactory，我们就不需要关心如何创建和更新 UUIDField 域了，全权交给 UUIDUpdateProcessorFactory 处理就行了。

```
<updateRequestProcessorChain name="dispup">
<processor class="solr.UUIDUpdateProcessorFactory">
<str name="fieldName">id</str>
</processor>
<processor class="solr.LogUpdateProcessorFactory" />
<processor class="solr.DistributedUpdateProcessorFactory" />
```

```

<processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
<requestHandler name="/update" class="solr.UpdateRequestHandler">
  <lst name="defaults">
    <str name="update.chain">dispup</str>
  </lst>
</requestHandler>

```

这样，我们创建索引的时候，就不用关心 id 域了，Solr 会自动帮我们创建并更新 id 域。但使用的时候需要注意的是，UUIDUpdateProcessorFactory 会判断你的 Document 中是否已经包含了 id 域，只有当 Document 不包含 id 域时，它才会自动帮你添加一个 id 域且域值是 UUID 字符串，内部其实就是通过 JDK 的 UUID 类来生成 UUID 字符串：

```
UUID.randomUUID().toString().toLowerCase(Locale.ROOT);
```

也就是说，一旦使用了 UUIDUpdateProcessorFactory，你就可以完全忘掉 id 域了，仿佛它根本不存在一样。当然使用 UUIDUpdateProcessorFactory 来自动管理 UUIDField 并不是必须的，也不是说有了 UUIDField，使用自增长的 ID 作为主键域就没有用武之地了。UUIDField 只是自增长 ID 域的一个备选替代方案。

## 7. 分词器的配置

在 FieldType 中，对于 TextField，我们还可以通过 <analyzer> 元素配置分词器，从而影响 TextField 的分词行为。下面是一个分词器配置示例：

```

<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt" ignoreCase=
"true" expand="false"/>
      -->
    <!-- Case insensitive stop word removal.
      -->
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"
    />
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
    <!-- Optionally you may want to use this less aggressive stemmer instead of
PorterStemFilterFactory:
    <filter class="solr.EnglishMinimalStemFilterFactory"/>
      -->
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
  <analyzer type="query">

```

```
<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase=
"true" expand="true"/>
<filter class="solr.StopFilterFactory"
    ignoreCase="true"
    words="lang/stopwords_en.txt"
/>
<filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
<filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
    <!-- Optionally you may want to use this less aggressive stemmer instead of
PorterStemFilterFactory:
    <filter class="solr.EnglishMinimalStemFilterFactory"/>
    -->
<filter class="solr.PorterStemFilterFactory"/>
</analyzer>
</fieldType>
```

<analyzer> 元素表示一个分词器，它有个 type 属性，可选的属性值有 index、query。index 表示索引时进行分词，query 表示在查询时对用户输入的关键词进行分词。<analyzer> 元素下又可以配置一个 <tokenizer> 加多个 <filter>，通过这种 tokenizer 与 filter 的不同组合实现不同的分词效果，你也可以实现自己的 tokenizer 或 filter。当然也可以把 tokenizer 与 filter 这种类似搭积木的组合行为隐藏在 analyzer 分词器实现类的代码内部，然后直接在 <analyzer> 元素的 class 属性配置上分词器类的完整包路径即可，这样配置分词器配置看起来更清爽了许多，但并不推荐这样做，毕竟把 tokenzier+filter 这种组合行为隐藏在代码里，不便于修改，因为你必须修改源码然后重新打成 jar 包。

3.3.6 Solr 的域

Solr 中的域用来表示 Document 的某一部分信息，而一个 Document 可能由多个域组成。Solr 的域需要在 schema.xml 中定义。只要你定义好了域类型，定义域就变得很简单了。下面是一个 Field 的简单配置示例：

```
<field name="price" type="float" default="0.0" indexed="true" stored="true"/>
```

上面示例中配置了一个名称为 price 的域，它的域类型为 float，默认值为 0.0，indexed=true 表示需要为 price 域创建索引，stored=true 表示需要存储 price 域的域值。

Field 拥有如表 3-3 所示的配置属性。

表 3-3 Field 配置属性

属性	描 述
name	域的名称，每个域都需要一个名称，且域的名称需要在同一个 Schema 文件中保持唯一性
type	域类型的名称，需要能够在 <fieldType> 的 name 属性中找到
default	域的默认值，当创建索引时，若域的域值没有赋值，那么默认值就派上用场了

Field 与 FieldType 拥有很多相同的配置属性，表 3-4 列出的属性既可以在某个特定的 Field 上指定也可以在某个特定的 FieldType 上指定，若两者同时指定了同一个配置属性，那么 Field 会覆盖 FieldType 的配置，若 FieldType 没有显式的指定同一个配置属性，Field 仍然会覆盖 FieldType 底层隐式设置的同名配置属性。

表 3-4 Field 与 FieldType 共同的可选配置属性

属性	描 述
indexed	表示是否需要索引，若配置为 true，域才能被查询，默认值 true
stored	表示是否需要存储，若配置为 true，域的原始值才能被提取，如果你需要使用 Highlight 或 MoreLikeThis 功能，那么 stored 必须设置 true，默认值 true
compressed	表示是否对域值进行压缩，只有 stored=true 时 compressed 设置为 true 才有意义，且 compressed 属性值只适用于 TextField 和 StrField，默认为 false
docValues	若配置为 true，那么域值会被放入一个 column-doc 的 docValues 结构中，对 facet 查询，group 分组，排序，function 查询有好处，尽管这个属性不是必须的，但他能加快索引数据加载，对 NRT 近实时搜索比较友好，且更节省内存，但它也有一些限制，比如当前 docValues 域只支持 strField,UUIDField,Trie*Field 等域，且要求域的域值是单值不能是多值域，默认值为 false
sortMissingFirst sortMissingLast	表示当一个排序域的值为 NULL 时，该如何对齐排序， sortMissingFirst 表示排在最前面， sortMissingLast 表示排在最后面， 默认值都是 false
multiValued	表示当前域是否为多值域，默认值 false
omitNorms	此属性若设置为 true，即表示将忽略域值的标准化因子，忽略在索引过程中对当前域的权重设置，且会节省内存。只有全文本域或者你需要在索引创建过程中设置域的权重时才需要把这个值设为 false，对于基本数据类型且不分词的域，如 intField,longField,StrField 等默认此属性值就是 true，否则默认就是 false。
omitTermFreqAndPositions	表示是否忽略 Term 的频率、Term 的位置信息、Term 的 payload 信息
omitPositions	跟 omitTermFreqAndPositions 类似，但 omitPositions 会保存 Term 的频率信息
termVectors termPositions termOffsets termPayloads	表示是否启用 Term 的空间向量模型； 表示是否保存 Term 的 position 位置信息； 表示是否保存 Term 的偏移量； 表示是否保存 Term 的 payload 信息； 存储这些额外信息会增大索引体积
required	表示某个域对于 document 是否是必需的，若设置为 true，但 Document 中却不存在该域，那么索引添加会失败。默认值 false
useDocValuesAsStored	若配置为 true，则会将该域当作 Stored Field 即便它的 stored 设置为 false，前提是 docValues 设置为 true 此配置才有效

## 1. Solr 的复制域

有时候，某个域的域值可能已经在其他域中存储，你可能不希望再重复提取数据并赋值，而是期望能不能从其他域上复制过来。而 Solr 的复制域就能解决这类问题。

这是一个 CopyField 的简单配置示例，代码如下所示：

```
<copyField source="cat" dest="text" maxChars="30000" />
<copyField source="vct" dest="text" maxChars="30000" />
```

在这个示例中，text 域的域值是从 cat 和 vct 这两个域复制而来的，source 表示复制的源域名称，dest 表示复制的目标域名称。source 和 dest 的属性值都支持通配符。maxChars 表示最多从 source 复制多少个字符到 dest。正如示例所示，你可以把多个域的值复制到一个目标域上即多次复制。需要注意的是，如果只复制单个域且被复制域是多值域，那么目标域也是多值域，这毋庸置疑，如果复制的是多个域，只要其中有一个域是多值域，那么目标域就一定是多值域，这点一定要谨记。

CopyField（复制域）的一个典型应用场景就是创建一个单一的搜索域来满足用户的各种查询需求。举个例子，用户搜索一本书籍时，title、author、keywrods、description 这几个域都应该查询，但我们设计搜索功能的时候，又不能强制用户指定在哪些域上进行搜索，即搜索域对于用户来说是完全透明的，此时我们可以定义一个 copyField，将 title、author、keywrods、description 这几个域的域值全部 copy 到新定义的 copyField 上，之后我们就可以设置默认搜索都对这个 copeField 进行查询。不过需要注意一点的是，使用 CopyField 会增大你的索引体积，增大索引体积是否会带来问题，这取决于你复制了多少个域到目标域上，以及被复制的源域的域值大小，域的分词情况，甚至你硬盘空间的使用状况。

对于同一个数据，我们有时候可能需要定义多个不同的域来应付不同的需求，举个例子，比如你有一个作者域 author，作者数据是：

Jack, Hunter, Herbert, Lewis

为了满足查询需求，你可能需要创建一个 seachField，seachField 的域值需要从原始的 author 域上复制而来，然后对 seachField 索引并分词。

为了满足排序需求，你可能又需要创建一个 sortField，sortField 的域值也需要从原始的 author 域上复制而来，但 sortField 不需要分词。

为了满足 Facet 查询需求，你可能又需要创建一个 facetField，facetField 的域值或许只是 author（因为 author 域可能是一个多值域，因为一本书的作者可以有多个）域的部分值，因为你可以并不希望统计所有的作者，所以 facetField 的域值可以不用从 author 域复制。

## 2. Solr 的动态域

Solr 的动态域允许你在为域创建索引时不再需要显示的在 schema.xml 中定义域，当你创建索引文档但忘记定义域时，如果没有动态域，那么索引创建一定会失败，但如果有了动态域，它使得你的程序不再那么脆弱。况且业务需求说变就变已经司空见惯，因此你可能会经常需要往 schema.xml 里添加新的域，但添加了新的域，你的 schema.xml 需要频繁的重新加载，这简直就是噩梦，如果有了动态域，则可以一定程度上减轻业务需求变化给 Solr 带来的影响，动态域的出现为 Solr 带来了一些灵活性。

动态域的 name 属性支持模糊匹配，它需要为 name 属性指定一个通配符表达式，当你索引文档时，如果一个 Field 找不到，那么就会尝试根据通配符去匹配动态域。



举个例子，假设你的 schema.xml 中定义了一个 name="\*\_i" 的动态域，当你试图索引一个包含 cost\_i 的域时，但是 schema.xml 中并没有显式的定义 name="cost\_i" 的域，于是 Solr 就会从定义的 dynamicField 动态域里去查找，依据 dynamicField 的 name 通配符去匹配。下面是一个 dynamicField 的简单配置示例，如下代码所示：

```
<dynamicField name="*_i" type="int" indexed="true" stored="true"/>
<dynamicField name="*_is" type="int" indexed="true" stored="true" multiValued="true"/>
```

### 3. 其他 Schema 元素

```
<uniqueKey>id</uniqueKey>
```

uniqueKey 元素，它用来配置 Document 的唯一标识域，即 Solr 是使用此域来决定增量导入时是否重复导入，如果 id 一样，则不会重复导入，或者当你更新索引时，你可以根据指定的 uniqueKey 域，来确定唯一一个 Document，然后对该 Document 进行更新。总之，它是用来一个确定唯一 Document 的，跟数据库表里的主键 id 概念类似，前提是 uniqueKey 里配置的域名称，你需要提前使用 field 元素进行定义。



**注意** CopyField 不能作为 uniqueKey。

```
<defaultSearchField>text</defaultSearchField>
```

用于配置默认搜索域，虽然已经提示此配置已经过时了，但 Solr 5 中仍然支持此配置。在 Solr 5.x 中，推荐你使用 df 参数来代替，因为这些配置可能会在后续版本中被移除并且不再支持。

```
<similarity class="solr.BM25Similarity"/>
<similarity class="solr.SchemaSimilarityFactory">
  <str name="defaultSimFromFieldType">text_dfr</str>
</similarity>
<fieldType name="text_dfr" class="solr.TextField">
  <analyzer ... />
<similarity class="solr.DFRSimilarityFactory">
  <str name="basicModel">I(F)</str>
  <str name="afterEffect">B</str>
  <str name="normalization">H3</str>
  <float name="mu">900</float>
</similarity>
</fieldType>
```

用于配置索引文档评分器，<similarity> 既可以配置在 <schema> 元素下，也可以配置在某个 <fieldType> 下，配置在 <schema> 元素下，则是全局有效，而配置在某个 <fieldType> 下，则只会影响应用了该 FieldType 的所有域。

```
<solrQueryParser defaultOperator="OR"/>
```

配置 Solr 的 QueryParser 默认使用的操作符为 OR，不配置默认就是 OR。这个配置已经过时了，后续版本中可能会被移除，所以不建议使用，推荐你使用 q.op 请求参数来代替。

### 3.3.7 Schema API

Schema API 为每个 Collection 或 Core 提供了读写访问 Solr Schema 的接口。所有 Schema 元素都可以被读取访问，Field、CopyField、DynamicField、FieldType 这些元素可以通过 Schema API 添加、删除、更新。未来 Solr 的正式发布版中可能会支持对更多 Schema 元素的写入访问。

如果修改了某个域的域类型，你可能需要重建所有索引数据，如果没有重建索引，你可能无法访问索引文档。修改 schema.xml 文件并不会影响已经索引了的文档，但想要应用 schema.xml 的修改，你需要重建索引。

Solr 的 Schema API 允许远程客户端访问 Schema 信息以及通过 REST 接口来更新 Schema 信息。Schema API 支持读取 Schema 的所有类型数据，但修改 Schema 信息需要你配置支持 Schema 可管理并且 Schema 可变的 SchemaFactory：ManagedIndexSchemaFactory。在你的 solrconfig.xml 中配置 ManagedIndexSchemaFactory 的示例如下所示：

```
<schemaFactory class="ManagedIndexSchemaFactory">
  <bool name="mutable">true</bool>
  <str name="managedSchemaResourceName">managed-schema</str>
</schemaFactory>
```

ManagedIndexSchemaFactory 有以下两个配置属性：

- ❑ mutable：控制是否允许运行时修改 Schema 信息，如果你想要启用 Schema API，mutable 必须设置为 true，如果设置为 false，即表示锁住 Schema 文件禁止修改；
- ❑ managedSchemaResourceName：定义 ManagedIndexSchemaFactory 管理的 schema 配置文件名称，若不指定默认是 managed-schema。若 <schemaFactory/> 没有在 solrconfig.xml 中显式指定，Solr 会隐式的使用 ManagedIndexSchemaFactory。

不过，管理 Schema 的另一种可选方式就是显式的在 solrconfig.xml 中配置 ClassicIndexSchemaFactory，ClassicIndexSchemaFactory 要求使用 schema.xml 配置文件，但它不支持运行时修改 Schema 信息，schema.xml 配置文件只能通过手动修改，且想要修改立即生效，需要重新加载 Core 或 Collection，或者重启你的 Solr Server。官方给出的 solrconfig.xml 示例文件中默认配置的就是 ClassicIndexSchemaFactory 即默认 Schema API 没有启用。

如果你之前使用的是 ClassicIndexSchemaFactory，现在你想切换到 ManagedIndexSchemaFactory，你可以修改 solrconfig.xml 中的 <schemaFactory/> 配置，示例上面已经给出。当 Solr 被重启，Solr 会检测 managedSchemaResourceName 属性配置的 Schema 文件（默认文件名为 managed-schema.xml）是否存在，若 managed-schema.xml 文件不存在，但 schema.xml 文件存

在，那么 Solr 会将 schema.xml 文件重命名为 schema.xml.bak，然后新建 managed-schema.xml 文件并将 schema.xml 文件内容写入到 managed-schema.xml 文件中，查看 managed-schema.xml 文件的头部，你会看到如下信息：

```
<!-- Solr managed schema - automatically generated - DO NOT EDIT -->
```

然后你就可以使用 Schema API 来管理你的 Schema 配置文件。

有时候可能又想切换回手动修改 Schema 配置文件，你可能需要以下步骤：

- ❑ managed-schema.xml 重命名为 schema.xml；
- ❑ 修改 solrconfig.xml 配置文件中的 <schemaFactory/> 元素的 class 属性值为 ClassicIndex-SchemaFactory；
- ❑ 重新加载你的 Core。

如果你正在使用 SolrCloud，那么可能需要借助 Zookeeper 来修改配置文件。

对于所有对 Schema API 的调用，返回的输出模型有两种：JSON 和 XML。当调用 Schema API 去修改 Schema 信息时，Solr Core 会自动重新加载，以便这些 Schema 修改能立即对 Schema 修改之后添加的索引文档有效。对于 Schema 信息修改之前已经索引的文档，你只能重建索引。

### 1. Schema API 入口点

Schema API 接口的 baseURL 为：

```
http://<host>:<port>/solr/<collection_name>
```

(1) /schema

查询或更新 Schema 信息，比如添加删除替换域、动态域、复制域、域类型。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema?wt=json
```

wt 参数可选值有 json/xml/schema.xml。

(2) /schema/fields

获取 schema.xml 中定义的所有域的信息或者获取指定域的信息。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema/fields
GET http://localhost:8983/solr/gettingstarted/schema/fields/fieldname
```

fieldname 表示指定的域名称。

支持的请求参数如表 3-5 所示。

表 3-5 请求参数

参数名	描述	类型	是否必需	默认值
wt	响应结果输出格式，可选值 json/XML	string	否	json

(续)

参数名	描述	类型	是否必需	默认值
fl	返回哪些域的信息，多个域名称用逗号分隔	string	否	所有域
includeDynamic	响应结果集里是否包含动态域	boolean	否	false
showDefaults	响应结果集里是否包含域的默认属性信息	boolean	否	false

### (3) /schema/dynamicfields

获取 schema.xml 中定义的所有动态域的信息或者获取指定动态域的信息。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema/dynamicfields
GET http://localhost:8983/solr/gettingstarted/schema/dynamicfields/fieldname
```

fieldname 表示指定的动态域名称。

支持的请求参数如表 3-6 所示。

表 3-6 请求参数

参数名	描述	类型	是否必需	默认值
wt	响应结果输出格式，可选值 json/XML	string	否	json
showDefaults	响应结果集里是否包含动态域的默认属性信息	boolean	否	false

### (4) /schema/fieldtypes

获取 schema.xml 中定义的所有域类型的信息或者获取指定域类型的信息。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema/fieldtypes
GET http://localhost:8983/solr/gettingstarted/schema/fieldtypes/fieldtypename
```

fieldtypename 表示指定的域类型名称。

支持的请求参数如表 3-7 所示。

表 3-7 请求参数

参数名	描述	类型	是否必需	默认值
wt	响应结果输出格式，可选值 json/XML	string	否	json
showDefaults	响应结果集里是否包含域类型的默认属性信息	boolean	否	false

### (5) /schema/copyfields

获取 schema.xml 中定义的所有复制域的信息。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema/copyfields
```

支持的请求参数如表 3-8 所示。

表 3-8 请求参数

参数名	描述	类型	是否必需	默认值
wt	响应结果输出格式，可选值 json/XML	string	否	json
source.fl	返回哪些 source 域的信息，多个域名称用逗号分隔	string	否	所有 source 域
dest.fl	返回哪些 dest 域的信息，多个域名称用逗号分隔	string	否	所有 dest 域

(6) /schema/name

获取 schema.xml 中定义的 name 信息。

示例如下：

```
GET http://localhost:8983/solr/gettingstarted/schema/name
```

(7) /schema/version

获取 schema.xml 的版本号。

(8) /schema/uniquekey

获取 schema.xml 中定义的 uniqueKey 信息。

(9) /schema/similarity

获取 schema.xml 中定义的全局 similarity 评分器信息。

(10) /schema/solrqueryparser/defaultoperator

获取 solrqueryparser 的默认操作符。

2. 修改 Schema 接口

接口 URL：

POST /collection/schema

通过此接口你可以添加、删除、替换域、动态域、复制域、域类型信息，此接口支持下列操作命令：

(1) add-field

添加一个域到 Schema 配置文件中，如果域名称重复了，会抛出异常，发送的 request payload 格式如下所示：

```
{
  "add-field":{
    "name":"sell-by",
    "type":"tdate",
    "stored":true }
}
```

(2) delete-field

删除一个域，如果该域不存在，或者该域是复制域的 source 域或 dest 域，那么会抛出异常，发送的 request payload 格式如下所示：

```
{"delete-field" : { "name":"sell-by" }}
```

### (3) replace-field

使用新的配置替换一个已存在域的旧配置信息，如果该域不存在，那么会抛出异常，发送的 request payload 格式如下所示：

```
{
  "replace-field":{
    "name":"sell-by",
    "type":"date",
    "stored":false
  }
}
```

### (4) add-dynamic-field

添加一个动态域到 Schema 配置文件中，如果动态域名称重复了，会抛出异常，发送的 request payload 格式如下所示：

```
{
  "add-dynamic-field":
  {
    "name":"*_s",
    "type":"string",
    "stored":true
  }
}
```

### (5) delete-dynamic-field

删除一个动态域，如果该动态域不存在，或者该动态域是复制域的 source 域或 dest 域，那么会抛出异常，发送的 request payload 格式如下所示：

```
{
  "delete-dynamic-field":{
    "name":"*_s"
  }
}
```

### (6) replace-dynamic-field

使用新的配置替换一个已存在动态域的旧配置信息，如果该动态域不存在，那么会抛出异常，发送的 request payload 格式如下所示：

```
{
  "replace-dynamic-field":{
    "name":"*_s",
    "type":"text_general",
    "stored":false
  }
}
```

### (7) add-field-type

添加一个域类型到 Schema 配置文件中，如果域类型名称重复了，会抛出异常，发送的



request payload 格式如下所示:

```
{
  "add-field-type" : {
    "name": "myNewTxtField",
    "class": "solr.TextField",
    "positionIncrementGap": "100",
    "analyzer" : {
      "charFilters": [{
        "class": "solr.PatternReplaceCharFilterFactory",
        "replacement": "$1$1",
        "pattern": "([a-zA-Z])\\\\\\\\1+" }],
      "tokenizer": {
        "class": "solr.WhitespaceTokenizerFactory" },
      "filters": [{
        "class": "solr.WordDelimiterFilterFactory",
        "preserveOriginal": "0" } ]}]
  }
}
```

#### (8) delete-field-type

删除一个域类型, 如果该域类型不存在, 或者存在任何普通域或动态域应用了该域类型, 那么会抛出异常, 发送的 request payload 格式如下所示:

```
{
  "delete-field-type": { "name": "myNewTxtField" }
}
```

#### (9) replace-field-type

使用新的配置替换一个已存在域类型的旧配置信息, 如果该域类型不存在, 那么会抛出异常, 发送的 request payload 格式如下所示:

```
{
  "replace-field-type": {
    "name": "myNewTxtField",
    "class": "solr.TextField",
    "positionIncrementGap": "100",
    "analyzer": {
      "tokenizer": {
        "class": "solr.StandardTokenizerFactory" } }
  }
}
```

#### (10) add-copy-field

添加一个复制域到 Schema 配置文件中, 发送的 request payload 格式如下所示:

```
{
  "add-copy-field": {
    "source": "shelf",
    "dest": [ "location", "catchall" ],
    "maxChars": "150"
  }
}
```

```
}
}
```

### (11) delete-copy-field

删除一个复制域，如果该复制域不存在，则会抛出异常，发送的 request payload 格式如下所示：

```
{
  "delete-copy-field":{"source":"shelf", "dest":"location"}
}
```

当使用 Schema API 修改 Schema 信息，Solr Core 会自动重新加载以便 Schema 信息的改变能立即应用于后续添加的索引文档，对 Schema 修改之前已经添加的索引文档无效。

你甚至可以在同一个 POST 请求里设置多个 Schema API 操作命令，并且同一个 POST 请求内的多个 Schema API 操作是具有事务特性的，也就是说要么所有 Schema API 操作成功，要么所有 Schema API 操作失败，这个 Schema API 操作严格按照他们定义的顺序被依次执行。这意味着如果你想要创建一个新的域类型并且在同一个请求内，那么创建域类型的命令必须放置在创建一个新域的前面，同理，一个域要想应用于一个复制域，那该域必须已存在，所以添加一个新域的命令应该放置在添加一个复制域的前面。

你可以在同一个 POST 请求里处理多个 Schema API 操作命令，具体示例如下所示：

```
{
  "add-field-type":{
    "name":"myNewTxtField",
    "class":"solr.TextField",
    "positionIncrementGap":"100",
    "analyzer":{
      "charFilters":[{
        "class":"solr.PatternReplaceCharFilterFactory",
        "replacement":"$1$1",
        "pattern":"([a-zA-Z])\\\\\\\\1+" }],
      "tokenizer":{
        "class":"solr.WhitespaceTokenizerFactory" },
      "filters":[{
        "class":"solr.WordDelimiterFilterFactory",
        "preserveOriginal":"0" }]]},
  "add-field" : {
    "name":"sell-by",
    "type":"myNewTxtField",
    "stored":true }
}
```

上面示例中演示了在同一个 POST 请求之内，执行了两个 Schema API 命令，先执行 add-field-type 添加了一个新的类型，再执行了 add-field 添加了一个新的域，由于 add-field-type 定义在前面，所以 add-field-type 比 add-field 先执行。

或者你也可以在同一个 POST 请求里，重复执行同一个 Schema API 命令，具体示例如下所示：

```
{
  "add-field":{
    "name":"shelf",
    "type":"myNewTxtField",
    "stored":true },
  "add-field":{
    "name":"location",
    "type":"myNewTxtField",
    "stored":true },
  "add-copy-field":{
    "source":"shelf",
    "dest":["location", "catchall" ]}
}
```

上面示例中，在同一个 POST 请求之内，先依次执行了 3 个 add-field 命令，最后执行了一个 add-copy-field 命令。对于连续重复的 Schema API 操作命令，你可以使用 Array 数组的形式发送，具体示例如下所示：

```
{
  "add-field":
  [
    {
      "name":"shelf",
      "type":"myNewTxtField",
      "stored":true
    },{
      "name":"location",
      "type":"myNewTxtField",
      "stored":true
    }
  ]
}
```

当你运行在 SolrCloud 模式下，你在一个节点上修改了 Schema 信息，会被广播到当前 collection 的所有副本上，你可以通过给请求传递 theupdateTimeoutSecs 参数（单位：秒），表示一直阻塞直到所有副本都应用到 Schema 的更新。这样可以使客户端程序更加的健壮，你可以确认所有副本在指定的时间内都确认了 Schema 更新。如果在指定的时间内，所有副本协议没有达成一致，那么客户端请求会失败，然后返回错误信息，错误信息里包含了是哪个副本发生问题。在大部分场景下，你只能选择在等待了短暂时间后再重试，如果重试后仍然出现问题，那么可能需要查阅在应用 Schema 更新时出现问题的副本所在 Server 的日志，如果你没有提供 updateTimeoutSecs 参数，那么默认行为是先保存 Schema 更新到 Zookeeper，然后立即返回给客户端接收节点，所有其他副本会异步应用 Schema 更新，因此，如果没有设置超时时间，你的客户端程序不能确保所有副本都应用到了最新的 Schema

更新。

### 3.3.8 Schemaless Mode

Schemaless Mode (无 Schema 模式) 是 Solr 的一个功能集合, 当这些功能一起使用时, 用户就可以快速的通过简单的索引数据构造一个高效的 Schema, 而不需要手动的去修改 Schema 文件。这些功能全部通过在 solrconfig.xml 文件里配置来完成控制。

- ❑ Managed schema : 配置 ManagedIndexSchemaFactory, 支持在运行时修改 Schema 信息;
- ❑ Field value class guessing: 域值的 Java 类型猜测;
- ❑ 基于域值的 Java 类型自动添加域到 Schema 里, 域值的 Java 类型会被映射到 Schema 的 FieldType 类型。

#### 1. 配置 Schemaless Mode

首先你需要在 solrconfig.xml 里配置 ManagedIndexSchemaFactory, 配置示例如下所示:

```
<schemaFactory class="ManagedIndexSchemaFactory">
  <bool name="mutable">true</bool>
  <str name="managedSchemaResourceName">managed-schema.xml</str>
</schemaFactory>
```

然后定义一个 UpdateRequestProcessorChain, 具体配置示例如下所示:

```
<updateRequestProcessorChain name="add-unknown-fields-to-the-schema">
  <!-- UUIDUpdateProcessorFactory will generate an id if none is present in
the incoming document -->
  <processor class="solr.UUIDUpdateProcessorFactory" />
  <processor class="solr.LogUpdateProcessorFactory"/>
  <processor class="solr.DistributedUpdateProcessorFactory"/>
  <processor class="solr.RemoveBlankFieldUpdateProcessorFactory"/>
  <processor class="solr.FieldNameMutatingUpdateProcessorFactory">
    <!-- 若域名里包含正则表达式匹配到的字符, 那么就将其替换成 replacement ->
    <str name="pattern">[^\w-\.]</str>
    <!-- 期望替换成的目标字符 ->
    <str name="replacement">_</str>
  </processor>
  <processor class="solr.ParseBooleanFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseLongFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseDoubleFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseDateFieldUpdateProcessorFactory">
    <arr name="format">
      <str>yyyy-MM-dd'T'HH:mm:ss.SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss,SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss.SSS</str>
      <str>yyyy-MM-dd'T'HH:mm:ss,SSS</str>
      <str>yyyy-MM-dd'T'HH:mm:ssZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss</str>
    </arr>
  </processor>
```

```

        <str>yyyy-MM-dd'T'HH:mmZ</str>
        <str>yyyy-MM-dd'T'HH:mm</str>
        <str>yyyy-MM-dd HH:mm:ss.SSSZ</str>
        <str>yyyy-MM-dd HH:mm:ss,SSSZ</str>
        <str>yyyy-MM-dd HH:mm:ss.SSS</str>
        <str>yyyy-MM-dd HH:mm:ss,SSS</str>
        <str>yyyy-MM-dd HH:mm:ssZ</str>
        <str>yyyy-MM-dd HH:mm:ss</str>
        <str>yyyy-MM-dd HH:mmZ</str>
        <str>yyyy-MM-dd HH:mm</str>
        <str>yyyy-MM-dd</str>
    </arr>
</processor>
<processor class="solr.AddSchemaFieldsUpdateProcessorFactory">
    <str name="defaultFieldType">strings</str>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Boolean</str>
        <str name="fieldType">booleans</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.util.Date</str>
        <str name="fieldType">tdates</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Long</str>
        <str name="valueClass">java.lang.Integer</str>
        <str name="fieldType">tlongs</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Number</str>
        <str name="fieldType">tdoubles</str>
    </lst>
</processor>
<processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>

```

上面示例中，UpdateRequestProcessorChain 允许 Solr 去猜测域值的类型，你需要预先定义默认的域类型，以及 FieldType 与 Java 数据类型之间的映射关系，即配置的 ParseXXXFieldUpdateProcessorFactory 的作用，然后自动添加域是 AddSchemaFieldsUpdateProcessorFactory 的作用。

Solr 内置提供了如下几种 ProcessorFactory：

- ❑ UUIDUpdateProcessorFactory：用于自动生成 UUID 域；
- ❑ RemoveBlankFieldUpdateProcessorFactory：用于自动移除域值为空的域；
- ❑ FieldNameMutatingUpdateProcessorFactory：用于自动根据提供的正则表达式修改域名称；
- ❑ ParseBooleanFieldUpdateProcessorFactory：用于自动转换成 Boolean 域；

- ❑ `ParseLongFieldUpdateProcessorFactory`: 用于自动转换成 Long 域;
- ❑ `ParseDoubleFieldUpdateProcessorFactory`: 用于自动转换成 Double 域;
- ❑ `ParseDateFieldUpdateProcessorFactory`: 用于自动转换成 Date 域;
- ❑ `AddSchemaFieldsUpdateProcessorFactory`: 用于自动添加域到 Schema。


## 2. Schemaless Mode 下创建索引

索引 CSV 格式数据, 示例如下所示:

```
接口 URL POST: http://localhost:8983/solr/gettingstarted/update?commit=true
Content-type:application/csv
// CSV 格式的索引数据
id,Artist,Album,Released,Rating,FromDistributor,Sold
44C,Old Shews,Mead for Walking,1988-08-13,0.01,14,0
```

与之前不同的是, 现在我们已经不需要提前在 `scehma.xml` 配置文件中定义 Field 啦! Solr 会自动根据提供的数据格式检测出应该选择哪个域类型, 并自动添加域。

---

 **注意** 即便你使用了 `schemaless mode` (无 Schema 模式), 你依然可以使用 Schema API 在运行时提前动态的创建域, 这些提前显式创建的域, 在索引文档时, 可以被使用。在 Solr 内部, Schma API 和 Schemaless Update Processors 都是使用的相同的 Managed Schema 功能。

---

## 3.4 data-config.xml 配置详解

`data-config.xml` 用于使用 Solr DIH 从外部数据库导入数据, 它的配置文件大致结构如下所示:

```
<dataConfig>
<dataSource name="mongoDataSource" ... />
<dataSource name="jdbcDataSource" ... />
<document>
<entity name="book"...>
<field column="bookName" name="bookName" mongoField="bookName"/>
<entity name="item"...>
<field column="lang" name="lang" />
</entity>
</entity>
</document>
<document>
<entity dataSource="jdbcDataSource" name="user" query="select * from user">
<field column="id" name="id"/>
<field column="user_name" name="userName"/>
...//
</entity>
```



```
</document>
</dataConfig>
```

如上所示, <dataconfig> 根元素下有 <dataSource> 和 <document> 两种元素, <dataSource> 元素表示数据源, 即需要导入到 Solr 中的数据来自哪里以及应该如何加载数据, 数据源可以配置多个。<document> 元素表示一个 Solr 索引, 其下可以定义多个 <entity> 元素, 每个 Entity 表示解析出来的一个实体, 它可能是数据库表里的一条记录, 也可能是文件里的一行数据等, Entity 元素下可以定义多个 <field>。每个 Entity 元素由多个 field 元素组成。Entity 元素还可以嵌套, 比如你有个 book 的 Entity, 其中有个表示书籍分类的 field: categoryID, 你可能希望对分类的名称进行索引, 而不是对分类的 id 数字进行索引, 此时你就可以定义一个子 Entity, 在子 Entity 里去加载 category 分类表获取分类名称信息。

data-config.xml 中还可以使用属性参数, 举个例子, 当你在配置 JDBC 数据源的时候, 你可能不希望将 jdbc url、driver、username、password 这些 JDBC 连接参数配置在 data-config.xml 文件里, 或许更希望能够单独配置在一个 properties 属性文件里, 然后通过参数名去替换, 这样方便统一管理这些参数, 因为这些参数在开发环境、测试环境、产品环境下不尽相同, 可能会需要经常来回切换, 使用 properties 属性文件显然更方便, 而 Solr 默认是支持的, 当你运行 DIH 导入数据时, Solr 默认会在 SOLR\_HOME/core/conf 目录下生成一个 dataimport.properties 属性文件, 当然也可以手动新建它, 你可以在其中添加自己需要的配置参数, 比如这样:

```
jdbcurl=jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-8
driver=com.mysql.jdbc.Driver
user=root
password=123
```

然后在 data-config.xml 中配置 JDBC 数据源时, 你就可以应用这些参数了, 示例如下所示:

```
<dataSource name="jdbcDataSource" type="JdbcDataSource" driver="${dataimporter.
request.driver}"
    url="${dataimporter.request.jdbcurl}"
    user="${dataimporter.request.user}"
    password="${dataimporter.request.password}"/>
```

自定义的参数引用格式为 \${dataimporter.request. 自定义参数名}, 对于 Solr 在 dataimporter.properties 属性文件中自动生成的参数比如 last\_index\_time, 引用的格式为 \${dih.last\_index\_time}。为了兼容 Solr 旧版本, \${dataimporter.last\_index\_time} 和 \${dataimporter.delta.id} 也是可以使用的。



**注意** \${dih.last\_index\_time} 替代了 \${dataimporter.last\_index\_time};  
\${dih.delta.id} 替代了 \${dataimporter.delta.id}。

## 3.5 zoo.cfg 配置详解

zoo.cfg 配置用于 SolrCloud 模式下，主要包含 Zookeeper 的一些运行参数，配置示例如下所示：

```
#ZK 中的时间配置最小单元，其他时间配置以整数倍 tickTime 来计算。
tickTime=2000
# Leader 允许 Follower 启动时在 initLimit 时间内完成数据同步，单位：tickTime
initLimit=10
# Leader 发送心跳包给集群中所有 Follower，若 Follower 在 syncLimit 时间内没有响应，
# 那么 Leader 就认为该 Follower 已经挂掉了，单位：tickTime
syncLimit=5
# 配置 ZK 的数据目录
dataDir=/usr/local/zookeeper/data
# 用于接收客户端请求的端口号
clientPort=2181
# 配置 ZK 的日志目录
dataLogDir=/usr/local/zookeeper/logs
#ZK 集群节点配置
# 端口号 2888 用于集群节点之间数据通信，端口号 3888 用于集群中 Leader 选举
server.1=192.168.123.100:2888:3888
server.2=192.168.123.101:2888:3888
server.3=192.168.123.102:2888:3888
```

关于 zoo.cfg 的详细说明，请查阅后续的 SolrCloud 章节，这里暂时略过。

## 3.6 本章总结

在本章中，我们主要详细了解了 Solr 中的各种配置文件，比如其中比较核心的 solrconfig.xml 和 schema.xml，以及数据导入时需要使用到的 data-config.xml。熟悉 Solr 配置能够扫清你在使用 Solr 过程中碰到的大部分障碍。希望大家不要强行记忆，要注重理解，并且在理解的基础上记忆配置。毕竟配置项繁多，理解才是灵活合理使用 Solr 配置的王道。

## Solr 分词

通过第 4 章，你将可以学习到以下内容：

- ☐ 分词的基本概念；
- ☐ 什么是 Analyzer；
- ☐ 什么是 Tokenizer；
- ☐ 什么是 TokenFilter；
- ☐ 学习并掌握 Solr 内置的 Tokenizer 和 TokenFilter 的使用；
- ☐ 学习并有选择性的掌握各种常见的中文分词器的使用。

接下来的章节将会讲解 Solr 是如何对原始文本进行分解和处理的，关于 Solr 分词，这里有 3 个主要概念需要理解：Analyzer，Tokenizer，TokenFilter。

### 4.1 分词的基本概念

分词就是将用户输入的一串文本分割成一个个 token，一个个 token 组成了 tokenStream，然后遍历 tokenStream 对其进行过滤操作，比如去除停用词、特殊字符、标点符号和统一转换成小写形式等。分词的准确与否，会直接影响搜索结果的相关度排序。从某种程度上来讲，分词算法的不同，都会影响页面的返回结果。因此，可以说分词是搜索的基础。

#### 4.1.1 理解 Analyzer

Analyzer 即分词器，用于将输入文本或文本输入流分解成 TokenStream 的工具。表面上看，貌似分词处理工作是由 Analyzer 完成的，其实 Analyzer 只不过是门脸，Analyzer

会通过 `createComponents` 函数创建一个 `TokenStreamComponents` 组件，`Analyzer` 内部是通过调用 `TokenStreamComponents` 的 `getTokenStream` 函数来生成 `tokenStream`。看起来貌似最终的 `tokenStream` 是由 `TokenStreamComponents` 生成的，在 `TokenStreamComponents` 内部维护了一个 `tokenizer` 和 `tokenStream`，而 `tokenStream` 又是 `tokenizer` 和 `tokenFilter` 的父类，但 `tokenizer` 存在，那么 `tokenStream` 只能是 `tokenFilter`，当 `tokenizer` 不存在，那么 `tokenStream` 可以做为 `tokenzier`，也可以作为 `tokenFilter`。因为 `tokenFilter` 有个 `TokenFilter (TokenStream input)` 的构造函数，很明显这里是个装饰者模式，也就是说 `tokenFilter` 既可以装饰 `tokenizer` 又可以装饰另一个 `toeknFilter`，这样就可以构成一个处理链条。而 `TokenStreamComponents` 只不过是这个处理链条的包装者，而 `TokenStreamComponents` 又被 `Analyzer` 包装。最终底层真正进行分词处理的是 `tokenizer` 和 `tokenFilter`，`tokenizer` 是分词处理的第一步，然后经过 `N` 个 `tokenFilter` 层层过滤，返回 `N` 个 `token`，而 `N` 个 `token` 又被 `tokenStream` 所包装，`tokenStream` 就好比 `token` 的 `Iterator` 迭代器，即分词器并不是直接返回 `N` 个 `token` 集合而是返回一个 `token` 的迭代器，通过 `tokenStream` 这个迭代器就能遍历得到所有 `token`。而 `token` 就是分词处理后的最小单元，也是索引创建的最小单元。对于外部用户来说，只需要使用 `Analyzer` 即可完成分词工作，不用关心内部工作原理。`Lucene` 分词器的整个结构流程图如图 4-1 所示。

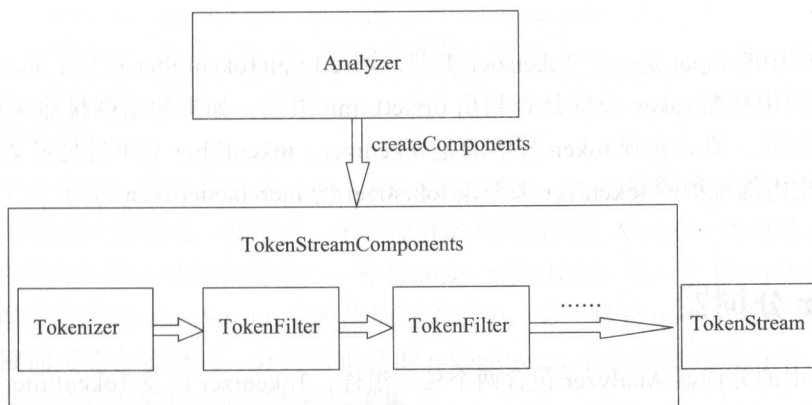


图 4-1 分词器的整个结构流程图

#### 4.1.2 理解 Tokenizer

`Tokenizer` 接收 `java.io.Reader` 作为输入，并通过 `incrementToken` 函数依次遍历每个 `token`，同时为每个 `token` 添加属性，比如 `postion`、`offset`、`payload` 等。因此 `Tokenizer` 才是 `token` 的生产者，它首先需要接收一个 `java.io.Reader` 字符输入流，而该 `Reader` 对象是通过 `TokenStreamComponents` 对象的 `setReader` 函数传递给 `tokenizer`，代码如下所示：

```
protected void setReader(final Reader reader) throws IOException {
    source.setReader(reader);
}
```

而 `TokenStreamComponents` 的 `Reader` 对象又是从何而来的呢？在 `Analyzer` 类的 `tokenStream` 函数中通过 `components.setReader(r)`；将 `reader` 对象射入到 `TokenStreamComponents` 中。对于 `Analyzer` 来说，`Reader` 无疑就是用户输入的待分词的文本内容了，如果用户输入的是 `String` 类型的字符串，那么 `Analyzer` 会将 `String` 转换成 `StringReader`。

### 4.1.3 理解 `TokenFilter`

`TokenFilter` 即 `Token` 过滤器，它内部维护了一个 `tokenizer`，通过调用 `tokenizer` 的 `incrementToken` 来完成 `token` 的遍历，定义过滤规则来完成 `token` 过滤。这是一个典型的装饰者模式。比如 `Solr` 默认自带的 `OffsetLimitTokenFilter` 实现，代码如下所示：

```
@Override
public boolean incrementToken() throws IOException {
    if (offsetCount < offsetLimit && input.incrementToken()) {
        int offsetLength = offsetAttrib.endOffset() - offsetAttrib.startOffset();
        offsetCount += offsetLength;
        return true;
    }
    return false;
}
```

上面代码中的 `input` 是一个 `Tokenizer` 类型，`OffsetLimitTokenFilter` 重写了 `incrementToken` 函数并通过调用判断 `token` 长度是否超出 `offsetLimit` 限制，如果超出则排除不符合要求的 `token`。由此可见，真正实现 `token` 分解的是 `tokenizer`，`tokenFilter` 只不过是定义一些过滤规则，然后调用内部维护的 `tokenizer` 来干预 `tokenizer` 的 `incrementToken` 行为。

## 4.2 Solr 分词器

`Lucene` 中的分词器 `Analyzer` 包含两个核心组件，`Tokenizer` 以及 `TokenFilter`。前者用于生成 `Token` 流，后者用于对 `tokenizer` 进行过滤。在 `Lucene` 中使用分词器可以直接创建相应的 `Analyzer` 对象，而在 `Solr` 中使用分词器，需要在 `schema.xml` 中配置 `<analyzer>` 元素，然后在 `<analyzer>` 元素下配置 `<tokenizer>` 和 `<filter>`，不过，`Solr` 里不能直接配置 `tokenizer`，而是由 `tokenizerFactory` 代替，即 `Tokenizer` 是由 `tokenizerFactory` 工厂创建的。当然你也可以直接配置一个 `<analyzer class="xxxAnalyzer">`。但这种直接配置 `Analyzer` 实现类的方式不推荐使用，建议通过 `<tokenizer>` 和 `<filter>` 元素动态组合的方式进行配置。想要在 `schema.xml` 中配置分词器，首先需要了解 `Lucene` 提供了哪些 `Tokenizer` 和 `TokenFilter`。需要说明的是，`Solr` 中并没有提供任何 `Tokenizer` 和 `TokenFilter` 的实现，`Solr` 中的自带分词器其实都是引用的 `Lucene` 项目。如今 `Solr` 与 `Lucene` 两个项目早已经合体，所以，提到 `Solr` 突然又跳到 `Lucene` 时，不要感到不适应，完全可以当两者是同一个项目。

### 4.2.1 Analyzer

通常情况下，只有 TextField 才需要指定 analyzer，配置 analyzer 最简单的方式就是使用 `<analyzer>` 元素，然后指定一个 class 属性，class 属性值设置为分词器类的完整包路径即可，Lucene 内置的分词器类都在 `org.apache.lucene.analysis.Analyzer` 包下，配置示例如下所示：

```
<fieldType name="nametext" class="solr.TextField">
  <analyzer class="org.apache.lucene.analysis.core.WhitespaceAnalyzer"/>
</fieldType>
```

在上面这个示例中，我们配置了一个 `WhitespaceAnalyzer` 分词器，它的职责是分析名称为 `nametext` 域的域值，以空格为分隔符进行分词，然后射出 `tokens`。这种简单的方式或许能应付大部分使用场景，但有时候，我们有更复杂的分词需求，通常更期望将分词处理分解成一个个独立的部件，以及一些相对简单的步骤。正如你看到的那样，Solr 默认已经提供了大量可选的 `Tokenizers` 和 `Tokenfilter` 配置示例，如下所示：

```
<fieldType name="nametext" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory"/>
    <filter class="solr.EnglishPorterFilterFactory"/>
  </analyzer>
</fieldType>
```

在这个示例中，我们没有为 `<analyzer>` 指定 class，而是提供了一系列的 `tokenizer` 和 `tokenfilter` 并将他们组合在一起形成一个处理链条共同扮演着 `Analyzer` 的角色。

分词在两种阶段下可能会发生，一种是 `index`（索引时），当一个 `Field` 被创建时，分词器分词处理后返回的 `tokenStream` 会被添加到索引中，另一种是 `query`（查询时），用户输入的查询关键词可能会被分词，分词后得到的 `tokenStream` 不会添加到索引中，查询时会与 `Field` 上的索引进行匹配从而返回查询结果集。

通常，一个相同的分词器可能会在两种阶段下被应用。当你希望索引阶段和查询阶段的分词处理有所不同，那么你可以配置两个 `<analyzer>`：`type="index"` 和 `type="query"`，配置示例如下所示：

```
<fieldType name="nametext" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
    <filter class="solr.SynonymFilterFactory" synonyms="syns.txt"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
```



```

    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>

```

从上面配置示例中可以看出，查询阶段的分词处理可以与查询阶段的分词处理不保持一致，但是一旦两者的分词行为没有保持一致，那么你在查询时，需要清楚的了解各自的分词行为，否则在查询返回结果时，你会搞不明白。

在某些查询中，比如前缀查询、正则查询，用户输入的查询关键词可能并不是自然语言，或许中间包含了通配符或者正则表达式，这时候我们不能按照常规的方式进行分词，比如用户输入的是 `an*`，假如我们还是按照常规的方式去配置分词，那么通配符可能会被当作特殊字符被剔除，这样就误解了用户的本意，最终的返回查询结果自然就差强人意，这不是我们所期望的查询效果。此时，Solr 提供另一种 `type = multiterm`，来应对类似前缀查询、正则查询这种特殊的 Multi-Term 查询，在 `multiterm` 场景下，用户输入的查询关键词是不会被分词的。下面是一个关于 `multi-term` 查询扩展的配置示例：

```

<fieldType name="nametext" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
    <filter class="solr.SynonymFilterFactory" synonyms="syns.txt"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="multiterm">
    <tokenizer class="solr.KeywordTokenizerFactory" />
  </analyzer>
</fieldType>

```

## 4.2.2 Tokenizer

前面我们已经知道，Tokenizer 的职责是用于分解生成 tokens 的，而 Analyzer 是由 1 个 Tokenizer 和 N 个 TokenFilter 组成的，而且 Tokenizer 是 analyzer 处理的第一个环节，Analyzer 必须要有一个 Tokenizer，而 TokenFilter 可以没有。需要注意的是，在 Solr 中，Tokenizer 并不能直接配置使用，需要通过 `tokenizerFactory` 工厂类来创建对应的 Tokenzier，即每个 Tokenizer 必须对应一个 `TokenizerFactory`。这也就意味着，假如你想要自定义一个 Tokenizer，除了需要继承 Tokenizer 基类，还需要继承 `TokenizerFactory` 实现与该 Tokenizer 对应的 `Tokenizer` 工厂类。`TokenizerFactory` 的职责就是创建一个特定的 Tokenizer，与 Tokenizer 一一对应。

下面是在 schema 中配置一个 `tokenizerFactory` 的简单示例：

```

<fieldType name="text" class="solr.TextField">

```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
</analyzer>
</fieldType>
```

Lucene 中默认提供了很多 Tokenizer 实现，在 Solr 中也提供了与之对应的 TokenizerFactory 实现，默认都是在该类名称后面加上 Factory。

### (1) KeywordTokenizer

KeywordTokenizer 会将整个文本当作一个 Token。它的工厂类是 KeywordTokenizerFactory。配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.KeywordTokenizerFactory"/>
</analyzer>
```

### (2) LetterTokenizer

LetterTokenizer 会提取出文本的所有连续的字母序列，并去除非字母字符，比如 “I can't” 会分成 “I”、“can”、“t”。

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.LetterTokenizerFactory"/>
</analyzer>
```

### (3) LowerCaseTokenizer

LowerCaseTokenizer 继承自 LetterTokenzier，所以 LowerCaseTokenizer 除了拥有 LetterTokenzier 的特性之外，还会将分成来的每个 Token 都转换成小写形式。比如 “I LOVE YOU” 会被分成 “i” “love” “you”。LowerCaseTokenizer 的工厂类是 LowerCaseTokenizerFactory。

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.LowerCaseTokenizerFactory"/>
</analyzer>
```


### (4) NGramTokenizer

根据给定的 GramRange 对域的文本生成 n-gram token。它的工厂类是 NGramTokenizerFactory。

NGramTokenizerFactory 有两个可选参数：

- 1) minGramSize: 最小 n-gram 大小，必须大于 0，默认值是 1；
- 2) maxGramSize: 最大 n-gram 大小，必须大于 0，默认值是 2。

---

 **注意** 必须 maxGramSize >= minGramSize，且 NGramTokenzier 不会按照空格进行分词，它会  
把域的文本当作一个整体进行 N-Gram 处理，因此，最后结果里，空格也会包含在内。

---

比如：

输入文本："hey man"

输出结果："h", "e", "y", " ", "m", "a", "n", "he", "ey", "y ", " m", "ma", "an"

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.NGramTokenizerFactory" minGramSize="2"
maxGramSize = "6" />
</analyzer>
```

### (5) EdgeNGramTokenizer

跟 NGramTokenizer 类似，不同的是 EdgeNGramTokenizer 每次都是从最左边或者最右边开始 n-gram。它的工厂类是 EdgeNGramTokenizerFactory。

EdgeNGramTokenizerFactory 有两个可选参数：

- 1) minGramSize: 最小 n-gram 大小，必须大于 0，默认值是 1；
- 2) maxGramSize: 最大 n-gram 大小，必须大于 0，默认值是 1。

Side：设置从哪个边界开始 n-gram，可选值有 front 和 back，front 表示从最前面开始，back 表示最后面开始即最右边，默认值是 front。

比如：输入文本："miss you"

minGramSize=2, maxGramSize=4

那么输出结果："miss", "miss ", "miss y", "miss yo", "miss you"

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.EdgeNGramTokenizerFactory" minGramSize="4" maxGramSize=
"8" />
</analyzer>
```

### (6) ICUTokenizer

ICUTokenizer 支持对多语言文本进行分词处理，比如汉语、英语、法语、德语、意大利语、韩语、日语等，当你需要分词的文本混杂了多个语种，那么 ICUTokenizer 是个不错的选择。它的工厂类是 ICUTokenizerFactory。

可选参数：

- 1) cjkAsWords: boolean 值，表示是否将中日韩字符当作一个英文单词，默认为 true；
- 2) rulefiles：表示一个 code:rulefiles 的键值对，code 表示 ISO-15924 脚本代号，rulefiles 表示一个 rbfi 资源文件路径。详细请自行搜索“ISO-15924”学习。


配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.ICUTokenizerFactory"
```

```
rulefiles="Cyr1:KeywordTokenizer.rbbi" />
</analyzer>
```

比如：输入：我是益达。123

输出："我"，"是"，"益"，"达"，"123"

 **注意** 使用 ICUTokenizer，你需要额外添加两个 jar 包：solr-5.3.1\contrib\analysis-extras\lib 下的 icu4j-version.jar 和 solr-5.3.1\contrib\analysis-extras\lucene-libs 下的 lucene-analyzers-icu-version.jar，将其添加到 \${SOLR\_HOME}\core\lib 目录下。

### (7) PathHierarchyTokenizer

PathHierarchyTokenizer 会根据给定的路径层次结构生成同义词。它的工厂类是 PathHierarchyTokenizerFactory，拥有如下可选参数：

- 1) delimiter：指定你的路径分隔符，比如反斜杠“/”；
- 2) replace：将 delimiter 字符使用 replace 表示的字符进行替换。

比如：

输入文本："c:\solrhome\core1\conf"

delimiter="\ " replace="/"

输出结果："c:"，"c:/ solrhome"，"c:/solrhome/core1"，"c:/solrhome/core1/conf"

配置示例如下所示：

```
<fieldType name="text_path" class="solr.TextField"
positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.PathHierarchyTokenizerFactory" delimiter="\ "
replace="/" />
  </analyzer>
</fieldType>
```

### (8) PatternTokenizer

PatternTokenizer 使用 Java 里的正则表达式来分割输入的文本并生成 Tokens。

Pattern 参数提供的正则表达式既可以理解为 token 的分隔符，也可以理解为匹配正则表达式的文本会被提取为 Token。它的工厂类是 PatternTokenizerFactory。

PatternTokenizerFactory 拥有如下配置参数：

- 1) pattern：Java 正则表达式，必需参数；
- 2) group：可选参数，默认值为 -1，用于指定提取哪个匹配组的文本作为 Token。

☐ group=-1 意味着 pattern 表示的正则表达式应该被当作 token 的定界分隔符；

☐ group 等于非负数表示指定的正则匹配组内的文本应该被当作 Token；

☐ group=0 表示匹配整个正则表达式；

□ `group>0` 表示匹配正则表达式指定分组的部分表达式，匹配组从左开始计算。  
配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern="\s*,\s*" />
</analyzer>
```

输入: "fee, fie, foe, fum, foo"

输出: "fee", "fie", "foe", "fum", "foo"

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory"
    pattern="[A-Z][A-Za-z]*" group="0" />
</analyzer>
```

输入: "Hello. My name is Yida"

输出: "Hello", "My", "Yida"

#### (9) UAX29URLEmailTokenizer

UAX29URLEmailTokenizer 支持从提供的文本中提取出 URL 和 Email 地址作为 Token。它会将空格和逗号连接符 - 作为 Token 的定界分隔符。UAX29URLEmailTokenizer 默认会按照空格和字符 “-” 作为 Token 的定界分隔符，比如 boy-friend 会被分成 boy 和 friend 两个 Token 而不是一个 Token。

UAX29URLEmailTokenizer 支持对以下格式的数据进行提取：

- 网络域名（包括顶级域名）；
- Email 地址；
- IP 地址，比如 192.168.134.100；
- file://、http(s)://、ftp:// 格式的 URL 链接。

UAX29URLEmailTokenizer 的工厂类是 UAX29URLEmailTokenizerFactory，它拥有一个可选的配置参数：

`maxTokenLength`：如果提取出来的 Token 的字符长度超过了 `maxTokenLength` 参数限定的大小，那么该 Token 会被丢弃。默认值 255。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.UAX29URLEmailTokenizerFactory"
    maxTokenLength="100" />
</analyzer>
```

比如：

输入: file:///C:/a/aa or ftp://a/as/b/ or http://www.jd.com or www.jd.com 192.166.56.12

Hi,girl,I can't,get over you,I wanna to be your boy-friend

输出: "file:///C:/a/aa","or","ftp://a/as/b/","or","http://www.jd.com","or",

"www.jd.com","192.166.56.12","Hi","girl","I","can't","get","over","you","I","wanna","to","be","your","boy","friend"

#### (10) WhitespaceTokenizer

WhitespaceTokenizer 会使用空白字符作为 Token 的定界分隔符来提取 Token。注意, 所有标点符号会包含在内。这里说的空白字符是使用 Character.isWhitespace(int) 方法来判断的。

WhitespaceTokenizer 的工厂类是 WhitespaceTokenizerFactory, 拥有如下可选配置参数:  
rule: 指定如何定义一个空白字符, 可选的参数值如下:

- 1) java: 默认值, 表示使用 Java 里的 Character.isWhitespace(int) 方法来原因;
- 2) unicode: 表示使用 UNICODE 的 WHITESPACE 属性。

配置示例如下:

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory" rule="java" />
</analyzer>
```

比如:

输入: " I miss you, I do "

输出: "I", "miss", "you", "I", "do"

#### (11) ClassicTokenizer

ClassicTokenizer 是一个使用 JFlex 基于语法构建的 Tokenizer。它适合于对欧洲语言的文档进行分词处理。ClassicTokenizer 先按照标点符号来切分 Token, 然后移除标点符号, 但是如果一个英文的句号后面没有跟一个空格, 那么这个句号也会作为 Token 的一部分。ClassicTokenizer 会按照连接符 "-" 切分 Token, 除非这个 Token 里包含数字 (比如这个 Token 可能是一个产品的编号 SKU-12), 否则这个 Token 会被中间的连接符 - 分割成两个 Token。ClassicTokenizer 还能识别 email 地址以及网络域名并生成 Token。

ClassicTokenizer 的工厂类是 ClassicTokenizerFactory, 拥有的可选配置参数如下:

maxTokenLength: 如果提取出来的 Token 的字符长度超过了 maxTokenLength 参数限定的大小, 那么该 Token 会被丢弃。默认值 255。

配置示例如下:

```
<fieldType name="text_classic" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.ClassicTokenizerFactory" maxTokenLength="100" />
  </analyzer>
</fieldType>
```

比如:

输入: SKU-12 123-456 SKU-II

输出: "SKU-12", "123-456", "SKU", "II"



### (12) StandardTokenizer

StandardTokenizer 是 Solr 官方提供的一个标准 Tokenizer 实现, 是基于 Unicode 文本分割算法实现的。注意, 它会按照逗号或者连接符 - 进行断词, 会剔除标点符号、# \$ % & 等这类特殊字符, 但不会剔除停用词, 不会转换大小写。

它的工厂类是 StandardTokenizerFactory, 拥有的可选配置参数如下:

**maxTokenLength:** 如果提取出来的 Token 的字符长度超过了 maxTokenLength 参数限定的大小, 那么该 Token 会被丢弃。默认值 255。

配置示例如下:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory" maxTokenLength="100" />
</analyzer>
```

### (13) SentenceTokenizer

SentenceTokenizer 会将输入的文本分解成一个个句子, 每个句子作为一个 Token, 短句的分隔符为中英文的逗号、分号、句号、感叹号、问号、顿号和一个以上的空格等。

它的工厂类是 SmartChineseSentenceTokenizerFactory, 没有配置参数。

配置示例如下:

```
<fieldType name="text_sentence" class="solr.TextField">
<analyzer>
<!-- 这里不能配置成 solr.SmartChineseSentenceTokenizerFactory -->
<tokenizer class="org.apache.lucene.analysis.cn.smart.
SmartChineseSentenceTokenizerFactory"/>
</analyzer>
</fieldType>
```

比如:

输入: Baby,I can't get over you,I wana to be your boy-friend.

输出: "Baby", "I can't get over you", "I wana to be your boy-friend."



**注意** 想要使用 SentenceTokenizer, 你需要复制 solr-5.3.1\contrib\analysis-extras\lucene-libs 目录下的 lucene-analyzers-smartcn-5.3.1.jar 到 \${SOLR\_HOME}/core/lib 目录下。

### (14) HMMChineseTokenizer

HMMChineseTokenizer 支持对中文或中英混合的文本进行分词。但对于中文只是进行简单的单字分割。分割 Token 时, 将 ! & % ^ ( ) \* \_ + - { | \ " ' ; , / < > @ ! ~ ` 等这些特殊字符当作 Token 的定界分隔符, 同时每个特殊字符添加一个字符内容为逗号 “,” 的 Token。

它的工厂类是 HMMChineseTokenizerFactory, 没有配置参数。

配置示例如下:

```
<fieldType name="text_sentence_hhmm" class="solr.TextField">
<analyzer>
<!-- 这里不能配置成 solr.HMMChineseTokenizerFactory-->
<tokenizer class="org.apache.lucene.analysis.cn.smart.
HMMChineseTokenizerFactory"/>
</analyzer>
</fieldType>
```

比如:

输入: 我是益达 !&yida

输出: "我", "是", "益", "达", ",", ",", ",", "yida"

HMMChineseTokenizerFactory 与 SmartChineseSentenceTokenizerFactory 在同一个 jar 包内, 所以你懂的。

开发阶段, 我们经常需要测试某个 Tokenizer 的分词效果。在 Lucene 里, 要达到这个目的, 我们需要自己编码实现, 比如这样:

```
/**
 * 用于分词器测试的一个简单工具类 (用于打印分词情况, 包括 Term 的起始位置和结束位置 (即所谓的
 * 偏移量),
 * 位置增量, Term 字符串, Term 字符串类型 (字符串 / 阿拉伯数字之类的))
 * @author Lanxiaowei
 *
 */
public class AnalyzerUtils {
    public static void main(String[] args) throws IOException {
        String text = "java <B>Hello</B> world";
        Analyzer analyzer = new BoldAnalyzer();
        displayTokens(analyzer, text);
    }

    public static void displayTokens(Analyzer analyzer, String text) throws
    IOException {
        TokenStream tokenStream = analyzer.tokenStream("text", text);
        displayTokens(tokenStream);
    }

    public static void displayTokens(TokenStream tokenStream) throws
    IOException {
        OffsetAttribute offsetAttribute =
tokenStream.addAttribute(OffsetAttribute.class);
        PositionIncrementAttribute positionIncrementAttribute =
tokenStream.addAttribute(PositionIncrementAttribute.class);
        CharTermAttribute charTermAttribute =
tokenStream.addAttribute(CharTermAttribute.class);
        TypeAttribute typeAttribute =
tokenStream.addAttribute(TypeAttribute.class);

        tokenStream.reset();
        int position = 0;
```

```

        while (tokenStream.incrementToken()) {
            int increment =
positionIncrementAttribute.getPositionIncrement();
            if (increment > 0) {
                position = position + increment;
                System.out.print(position + ":");
            }
            int startOffset = offsetAttribute.startOffset();
            int endOffset = offsetAttribute.endOffset();
            String term = charTermAttribute.toString();
            System.out.println "[" + term + "]" + ":" + (startOffset +
"-->" + endOffset + "):" + typeAttribute.type());
        }
    }

/**
 * 断言分词结果
 * @param analyzer
 * @param text 源字符串
 * @param expecteds 期望分词后结果
 * @throws IOException
 */
public static void assertAnalyzerTo(Analyzer analyzer, String
text, String[] expecteds) throws IOException {
    TokenStream tokenStream = analyzer.tokenStream("text", text);
    CharTermAttribute charTermAttribute =
tokenStream.addAttribute(CharTermAttribute.class);
    for (String expected : expecteds) {
        Assert.assertTrue(tokenStream.incrementToken());
        Assert.assertEquals(expected,
charTermAttribute.toString());
    }
    Assert.assertFalse(tokenStream.incrementToken());
    tokenStream.close();
}
}

```

但在 Solr 中，你不需要这么麻烦，你只需要在 schema.xml 中配置好 fieldType 或 field，然后在 Solr 的 Web 管理界面里，通过左侧 Analysis 菜单可以访问其提供的域类型或进行域的分词测试功能，如图 4-2 所示。

### 4.2.3 TokenFilter

TokenFilter 是 Tokenizer 的过滤器，它用于对 Tokenizer 处理后的 token 进行二次过滤。由于 TokenFilter 本身就是 Tokenizer 的子类，而 Analyzer 首先需要经过一个 Tokenizer，然后经过 N 个 TokenFilter 链条依次过滤。所以，Analyzer 其实可以没有 TokenFilter，直接经过 N 个 TokenFilter 也可以的，此时经过的第一个 TokenFilter 就充当了 Tokenizer 的角色，因为 TokenFilter 本身就是 Tokenizer 的子类，当然可以看作是一个 Tokenizer。只是全部使

用 TokenFilter 不太直观。

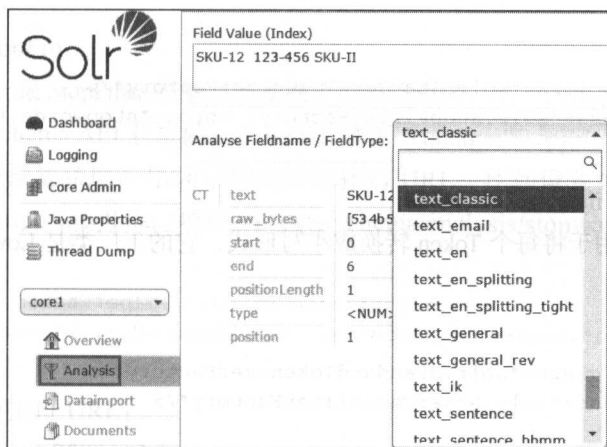


图 4-2 Solr 的分词测试功能

在 Solr 中, TokenFilter 都是在 schema.xml 配置文件中先声明的, 交给 Solr 去创建, 用户不用关心如何创建 TokenFilter, 何时创建 TokenFilter, 只需要明白如何正确配置 TokenFilter 即可。你只需要在 <analyzer> 元素下配置一个 <filter> 元素, <filter> 元素的 class 属性配置成 TokenFilter 的工厂类, 与 Tokenizer 类似, 每个 TokenFilter 都必须要有个 TokenFilterFactory 与之对应。假如你已经在 <analyzer> 元素下配置了 <tokenizer>, 那么 <filter> 元素必须配置在 <tokenizer> 之后, 因为最终会严格按照这里定义的顺序去依次执行。上面我们说过, 在 Analyzer 没有 Tokenizer 的情况下, 第一个 TokenFilter 就充当了 Tokenizer, 由此可知, <analyzer> 元素下可以没有 <tokenizer> 元素, 直接配置一个或多个 <tokenFilter> 元素也是可以的。当然, <analyzer> 元素下 <tokenizer> 元素和 <tokenFilter> 元素都不配置也是可以的, 此时你就必须为 <analyzer> 元素添加 class 属性, class 属性值为 Analyzer 分词器实现类的包路径, 对于 Solr 内置的 Analyzer 分词器, 包名可以用 solr. 前缀代替。

#### (1) StopFilter

StopFilter 会过滤掉在停用词字典里出现的 Token, 停用词字典文件默认名称为 stopwords.txt, 需要放置在 core/conf 目录下。StopFilter 的工厂类是 StopFilterFactory, 拥有如下可选参数:

- words: 可选参数, 用于配置停用词字典文件的加载路径, 既可以是绝对路径, 也可以是相对 core/conf 的路径。停用词字典文件的空白行和以 # 号开头的字符将会被忽略;
- format: 如果停用词列表已经被 Snowball 所格式化, 那么 format 可以设置为 snowball。可选参数;
- ignoreCase: 过滤停用词时是否忽略大小写, 默认为 false;
- enablePositionIncrements: Solr 4.3 版本之前的参数, 表示是否启用位置增量, 在

Solr 5.x 中已经移除此参数。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
</analyzer>
```

## (2) LowerCaseFilter

LowerCaseFilter 用于将每个 Token 转换成小写形式，它的工厂类是 LowerCaseFilterFactory。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
```

## (3) LengthFilter

LengthFilter 用于过滤掉不符合规定字符长度限制的 Token，即字符长度不在区间 [min,max] 内的 Token 就会被丢弃。它的工厂类是 LengthFilterFactory。LengthFilterFactory 拥有如下配置参数：

- ❑ min：token 字符的最小长度，小于这个值的 token 将会被丢弃，必需参数；
- ❑ max：字符的最大长度，大于这个值的 token 将会被丢弃，必需参数。注意，max >= min；
- ❑ enablePositionIncrements：Solr 4.3 版本之前的参数，表示是否启用位置增量，在 Solr 5.x 中已经移除此参数。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LengthFilterFactory" min="3" max="7"/>
</analyzer>
```

## (4) TrimFilter

TrimFilter 用于剔除 Token 前面和后面的空格字符，注意，它不会剔除 Token 中间包含的空格字符。它的工厂类是：TrimFilterFactory。

可选参数：

updateOffsets：这个参数是 Lucene 4.3 版本之前的参数，表示当移除空格字符后是否需要更新该 Token 的位置偏移量，对于 Lucene 5.x，此参数已经移除。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.PatternTokenizerFactory" pattern=","/>
```

```
<filter class="solr.TrimFilterFactory"/>
</analyzer>
```

### (5) ManagedStopFilter

ManagedStopFilter 跟 StopFilter 类似，都是用于过滤停用词，唯一区别就是 ManagedStopFilter 的停用词数据是从 Schema API 获取的。它的工厂类是 ManagedStopFilterFactory。

**managed**：用于设置获取停用词字典数据的接口 URL，比如设置为 english，那么完整的请求 URL 就是：<http://localhost:8080/solr/core/schema/analysis/stopwords/english>

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedStopFilterFactory" managed="english"/>
</analyzer>
```

访问停用词列表接口 URL：

GET <http://localhost:8080/solr/core/schema/analysis/stopwords/english>

返回的数据格式：

```
{
  "responseHeader":{
    "status":0,
    "QTime":1
  },
  "wordSet":{
    "initArgs":{"ignoreCase":true},
    "initializedOn":"2014-03-28T20:53:53.058Z",
    "managedList":[
      "a",
      "an",
      "and",
      "are",
      ... ]
  }
}
```

停用词数据都在 managedList 属性数组里。

添加停用词到停用词列表接口 URL：

PUT/POST <http://localhost:8080/solr/core/schema/analysis/stopwords/english>

Request payload 格式：["a", "an", "the", "is"]

测试某个停用词是否在停用词列表中存在的接口 URL：

GET <http://localhost:8080/solr/core/schema/analysis/stopwords/the>

直接在 URL 后面加上停用词即可，这里在 /stopwords/ 后面加上 the 表示判断停用词列表是否存在 the。

如果存在，则会返回状态码 200，否则就是不存在。

删除停用词列表里的某个停用词的接口 URL：



DELETEhttp://localhost:8080/solr/core/schema/analysis/stopwords/what

这里表示删除“what”这个停用词



**注意** 修改了停用词数据，不管修改的是停用词字典文件，还是通过 Solr 提供的 RESTFUL API 接口，你都需要重新加载你的 Core 才能使修改立即生效。

## (6) SynonymFilter

SynonymFilter 用于 Token 的同义词映射，同义词映射是需要提前在同义词字典文件里声明，同义词字典文件编写格式如下所示：

```
aaafoo => aaabar
bbbfoo => bbbfoo bbbbar bar
fooaaa,baraaa,bazaaa
GB,gib,gigabyte,gigabytes
ipod, i-pod, i_pod
Television, Televisions, TV, TVs
```

同义词字典文件的默认文件名称为 synonyms.txt，用户可以通过 synonyms 参数指定同义词字典文件的加载路径，既可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录。

SynonymFilter 的工厂类是 SynonymFilterFactory，SynonymFilterFactory 拥有如下配置参数：

- ❑ synonyms：用于配置同义词字典文件的加载路径，必需参数；
- ❑ ignoreCase：在匹配同义词时是否忽略大小写，默认值 false；
- ❑ expand：如果 expand=true，表示同一行的单词互为同义词，如果 expand=false，表示只有第一个单词与同一行的其他单词是同义词；
- ❑ format：用于控制如何解析同义词字典文件里的同义词，支持的实现有 Solr 实现的 SolrSynonymParser 以及 wordnet 的 WordnetSynonymParser，或者你也可以继承 SynonymMap.Builder 实现自己的 SynonymParser 同义词解析器，默认值是 Solr；
- ❑ tokenizerFactory：用于配置使用什么 tokenizerFactory 来对同义词字典文件内容进行分词从而提取同义词。默认值是 solr.WhitespaceTokenizerFactory。如果配置了这个参数，那么 analyzer 参数可以不用配置；
- ❑ analyzer：用于配置使用什么 Analyzer 类来对同义词字典文件内容进行分词从而提取同义词。默认值是 Solr.WhitespaceTokenizerFactory。

配置示例如下：

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
```

```
tokenizerFactory="solr.KeywordTokenizerFactory" expand="true"/>
</analyzer>
```

### (7) ManagedSynonymFilter

ManagedSynonymFilter 跟 ManagedStopFilter 类似，都是借助 Solr 提供的 RESTFUL API 接口来管理资源文件，唯一区别就是两者依赖的字典文件以及文件数据格式不同。它的工厂类是 ManagedSynonymFilterFactory。

managed：用于设置获取同义词字典数据的接口 URL，比如设置为 english，那么完整的请求 URL 就是：<http://localhost:8080/solr/core/schema/analysis/synonyms/english>

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedSynonymFilterFactory" managed="english"/>
</analyzer>
```

跟 ManagedStopFilterFactory 类似，同义词字典文件数据也可以通过 RESTFUL API 进行增删改查，接口 URL 也类似，只需要将接口请求 URL 里的 stopwords 替换成 synonyms 即可，接口其他方面说明请查看 ManagedStopFilterFactory 部分。

### (8) TypeTokenFilter

TypeTokenFilter 可以通过白名单或黑名单来过滤掉不符合规定的 Token 类型的 Token。Lucene 里支持的 Token 类型有以下几种：

- ❑ <ALPHANUM>：字母数字混合字符；
- ❑ <NUM>：纯数字；
- ❑ <URL>：URL 地址；
- ❑ <EMAIL>：Email 地址；
- ❑ <SOUTHEAST\_ASIAN>：南亚和东南亚语言的字符序列，包括泰国，老挝，缅甸，柬埔寨；
- ❑ <IDEOGRAPHIC>：中日韩的表意字符；
- ❑ <HIRAGANA>：日语里的单个平假名字符；
- ❑ <HANGUL>：韩语字符。

关于 TokenType，详情请查阅 StandardTokenizer 类的源码。

TypeTokenFilter 的工厂类是 TypeTokenFilterFactory，拥有如下配置参数：

- ❑ types：用于定义 TokenType 配置文件的加载路径，可以是绝对路径也可以是相对路径，相对路径是相对于 core/conf 目录；
- ❑ useWhitelist：如果设置为 true，那么 types 参数表示的就是白名单文件的加载路径，如果设置为 false，那么 types 参数表示的就是黑名单文件的加载路径。默认值 false；
- ❑ enablePositionIncrements：Solr 4.3 版本之前的参数，表示是否启用位置增量，在 Solr 5.x 中已经移除此参数。

配置示例如下所示：

```
<analyzer>
  <filter class="solr.TypeTokenFilterFactory" types="stotypes.txt" useWhitelist=
"true"/>
</analyzer>
```

stotypes.txt 配置示例如下所示：

```
<ALPHANUM>
<NUM>
<URL>
<EMAIL>
<IDEOGRAPHIC>
```

### (9) WordDelimiterFilter

WordDelimiterFilter 可以用于处理骆驼命名法的文本、帕斯卡命名法的文本、数字与非数字字符串混排的文本以及 A+ 连接符 +B 的混排文本 (A, B 可以是英文单词、数字、中文等), 比如是否需要分割成几部分, 分割成几部分后是否需要再合并在一起等。这里说的连接符指的是 ~!@#\$%^&\*( )\_+={}[]\” ’ ?/,.<> 等这些字符, 连接符可以是多个这些特殊字符的组合, 并不一定只是单个特殊字符作为连接符。同时 WordDelimiterFilter 支持删除英文里的人称代词所有格, 支持简单地按照空格分词。说这么多比较抽象, 还是以具体示例来说明 WordDelimiterFilter 到底能做什么吧!

- ❑ 骆驼命名法的文本: printEmployeePaychecks, 可以分成: printEmployeePaychecks;
- ❑ 帕斯卡命名法的文本: PrintEmployeePaychecks, 可以分成: PrintEmployeePaychecks;
- ❑ 英文 + 数字: SKU12, 可以分成: SKU 12;
- ❑ 数字 + 英文: 50KG, 可以分成: 50 KG;
- ❑ 数字 + 中文: 2006 届, 可以分成: 2006 届;
- ❑ 中文 + 数字: 我爱你 1314, 可以分成: 我爱你 1314;
- ❑ 数字 + 连接符 + 英文: 30-seconds, 可以分成: 30 seconds;
- ❑ 英文 + 连接符 + 数字: 2+DAYS, 可以分成: 2DAYS;
- ❑ 英文 + 连接符 + 英文: Wi-Fi, 可以分成: Wi Fi;
- ❑ 去除英文人称所有格: It's my treat, 可以分成: It my treat。

WordDelimiterFilter 的工厂类是 WordDelimiterFilterFactory, 拥有如下可选配置参数:

- ❑ generateWordParts: 可选参数, 表示分词时生成的非数字 Token 是否应该保留, 配置成 0 表示丢弃分出来的非数字 Token, 否则保留。默认值是 1 即会保留;
- ❑ generateNumberParts: 可选参数, 表示分词时生成的数字 Token 是否应该保留, 配置成 0 表示丢弃分出来的数字 Token, 否则保留。默认值是 1 即会保留;
- ❑ catenateWords: 可选参数, 当把文本分成 N 个部分后, 是否对相邻的两个非数字 Token 进行拼接, 比如 SKU-II, 会分成 SKU II SKUII。SKUII 就是 SKU 和 II 的拼

接。当如果是 SKU-123-II，那么最终就不会分出 SKU-II，因为 SKU 和 II 不相邻。如果 `catenateWords=0` 则表示不拼接，否则拼接。默认值是 0 即不拼接；

- ❑ `catenateNumbers`：可选参数，与 `catenateWords` 参数类似，表示是否对相邻的两个数字 Token 进行拼接，如果 `catenateNumbers=0` 则表示不拼接，否则拼接。默认值是 0 即不拼接；
- ❑ `catenateAll`：可选参数，是否表示把分词的 N 个部分全部拼接起来组成一个新的 Token。如果 `catenateAll` 配置为 0 则表示不全部拼接一起，否则全部拼接。默认值是 0 即不全部拼接；
- ❑ `splitOnCaseChange`：可选参数，表示对于骆驼命名法或帕斯卡命名法这类纯英文文本字符串，是否需要进行分割。如果 `splitOnCaseChange=0` 则表示不分割，否则会分割。默认值是 1 即会分割；
- ❑ `splitOnNumerics`：可选参数，表示对于英文 + 数字或者数字 + 英文这类文本字符串，是否需要进行分割。如果 `splitOnNumerics=0` 则表示不分割，否则会分割。默认值是 1 即会分割；
- ❑ `preserveOriginal`：可选参数，表示是否需要把分割前的原始文本保留下来单独作为一个 Token。如果 `preserveOriginal=0` 则表示不保留，否则会保留。默认值是 0 即不保留；
- ❑ `stemEnglishPossessive`：可选参数，表示是否需要剔除英文里的人称代词所有格，如果 `stemEnglishPossessive=0` 则表示不会剔除，否则会剔除。默认值是 1 即会剔除；
- ❑ `protected`：可选参数，用于配置受保护的词典文件的加载路径，添加到词典文件内的文本会受保护，不会被分割成 N 个 Token 且不会剔除人称代词所有格，即保留原样，词典文件路径可以是绝对路径也可以是相对路径，相对路径是相对 `core/conf` 目录；
- ❑ `types`：可选参数，表示字符 --> 类型的一个类型映射配置文件的加载路径，此配置文件用于指定某个字符属于什么类型，默认支持的类型有 LOWER, UPPER, ALPHA, DIGIT, ALPHANUM 和 SUBWORD\_DELIM，依次表示小写字母、大写字母、希腊的阿尔法字符、数字、阿尔法数字和分隔符。比如你可以在类型映射配置文件里配置 `%=>DIGIT` 即表示百分号 % 是一个数字类型，那么在进行 token 分割时，会把百分号当作数字处理。类型映射配置文件加载路径可以是绝对路径也可以是相对路径，相对路径是相对 `core/conf` 目录。

配置示例如下：

```
<fieldType name="text_worddelimiter" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory"
    generateWordParts="1"
    generateNumberParts="1"
```

```

        catenateWords="0"
        catenateNumbers="0"
        catenateAll="0"
        preserveOriginal="1"
        splitOnCaseChange="1"
        splitOnNumerics="1"
        types="wordtypes.txt" protected="protwords.txt" />
</filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
</fieldType>

```

字符类型映射配置文件配置示例如下：

```

$ => DIGIT
% => DIGIT
. => DIGIT
# 以 # 号开头的表示注释
#\u002C => DIGIT

```

保护词典文件配置示例如下：

```

# 以 # 号开头的表示注释
It's
iPhone7-Plus

```

#### (10) KeepWordFilter

KeepWordFilter 用于保护定义在字典文件里的 Token，使其保持原样，在字典里不存在的 Token 将会被丢弃。KeepWordFilter 与 StopFilter 很相似，StopFilter 与其刚好相反，是定义在词典文件里的所有 Token 将会被丢弃。

KeepWordFilter 的工厂类是 KeepWordFilterFactory，拥有如下配置参数：

- ❑ words：必需参数，用于指定字典文件的加载路径，每行配置一个受保护的词，空白行和以 # 号开头的行将会被忽略。字典文件加载路径可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录；
- ❑ ignoreCase：匹配受保护词时是否忽略大小写，默认值 false 即不忽略大小写；
- ❑ enablePositionIncrements：Solr 4.3 版本之前的参数，表示是否启用位置增量，在 Solr 5.x 中已经移除此参数。

配置示例如下：

```

<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.KeepWordFilterFactory" words="keepwords.txt"/>
</analyzer>

```

#### (11) EnglishPossessiveFilter

EnglishPossessiveFilter 用于剔除英文里的人称代词所有格，比如 's、' S、\' s、\' S、' s、' S、' s、' S 等，它的工厂类是 EnglishPossessiveFilterFactory，没有可选的配置参数。

配置示例如下：

```
<fieldType name="semicolonDelimited" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory" />
    <filter class="solr.EnglishPossessiveFilterFactory" />
  </analyzer>
</fieldType>
```

## (12) SetKeywordMarkerFilter

SetKeywordMarkerFilter 是用于标记 Keyword 的过滤器，被标记为 Keyword 的 Token 会被额外添加一个 Keyword 属性。SetKeywordMarkerFilter 与 KeepWordFilter 有点类似，但不在字典文件里的 Token 或者不符合正则表达式的 Token 并不会被 SetKeywordMarkerFilter 剔除掉，SetKeywordMarkerFilter 只是为 Keyword 打上标记。它的工厂类是 KeywordMarkerFilterFactory，测试效果如图 4-3 所示。

可选参数：

- **protected**：keyword 字典文件加载路径，可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录。可选参数；
- **pattern**：用于匹配 Keyword 的正则表达式，符合此正则表达式的 Token 将会被额外添加一个 Keyword 的属性。可选参数，protected 与 pattern 参数至少要配置一个，如果两者都配置了，那么优先以 pattern 正则表达式为标准来匹配正则表达式。

配置示例如下：

```
<fieldType name="text_keywordmarker_pattern" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" pattern="^.+er$"/>
  </analyzer>
</fieldType>
```

或者使用 Keyword 字典文件来匹配 Keyword，每个 Keyword 独占一行。

```
<fieldType name="text_keywordmarker_pattern" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protectedkeyword.txt" />
  </analyzer>
</fieldType>
```

Keyword 字典文件配置示例如下所示：

```
Solr
Lucene
Java
Scala
```



Field Value (Index)		Field Value (Query)				
Solr Lucene Java Scala JavaScript						
Analyse Fieldname / FieldType: text_keywordmarker_pattern		<input checked="" type="checkbox"/> Verbose Output				
WT	text	Solr	Lucene	Java	Scala	JavaScript
	raw_bytes	[53 6f 6c 72]	[4c 75 63 65 6e 65]	[4a 61 76 61]	[53 63 61 6c 61]	[4a 61 76 61 53 63 72 69 70 74]
	start	0	5	12	17	23
	end	4	11	16	22	33
	positionLength	1	1	1	1	1
	type	word	word	word	word	word
	position	1	2	3	4	5
SKMF	text	Solr	Lucene	Java	Scala	JavaScript
	raw_bytes	[53 6f 6c 72]	[4c 75 63 65 6e 65]	[4a 61 76 61]	[53 63 61 6c 61]	[4a 61 76 61 53 63 72 69 70 74]
	keyword	true	true	true	true	false

图 4-3 KeywordMarkerFilterFactory 测试效果图

**注意** 当 pattern 参数配置了, 那么此时 tokenFilter 实现类不是 SetKeywordMarkerFilter, 而是 PatternKeywordMarkerFilter。PatternKeywordMarkerFilter 即通过指定的正则表达式来匹配 Keyword 并为其打上 Keyword 标记, 两者共用同一个 KeywordMarkerFilterFactory。

(13) PorterStemFilter

PorterStemFilter 应用 Porter Stemming 算法对英文进行词干还原, 但它只能进行简单的词干还原, 比如去除单词结尾的 ed/s/ing 等等, 但类似 thought → think、took → take、broken → break、done → do 这类的词干还原, PorterStemFilter 并不支持。它的工厂类是 PorterStemFilterFactory。

配置示例如下:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory" />
  <filter class="solr.PorterStemFilterFactory" />
</analyzer>
```

(14) EnglishMinimalStemFilter

EnglishMinimalStemFilter 用于将英文单词复数形式转换成单数形式, 比如 books → book, heroes → hero, cities → city, 但对于类似 boxes 这种以 es 结尾的复数形式不支持, 目前只支持以 s 结尾或 ies 结尾的单词复数形式转换成单数形式!

它的工厂类是 EnglishMinimalStemFilterFactory, 没有可选参数。

配置示例如下:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory" />
```

```
<filter class="solr.EnglishMinimalStemFilterFactory"/>
</analyzer>
```

### (15) PatternReplaceCharFilter

PatternReplaceCharFilter 用于将指定正则表达式与每个 Token 进行匹配, 如果有匹配到, 那么将匹配到的部分替换成指定的字符串, 但这个功能可能会影响高亮功能的使用, 因为替换后 Token 的位置偏移量与之前的原始位置偏移量可能不一致, 从而导致高亮标记错位。PatternReplaceCharFilter 的工厂类是 PatternReplaceCharFilterFactory, 拥有如下配置参数:

- pattern: Java 正则表达式, 必需参数;
- replacement: 替换字符, 可选参数, 如果没有指定, 那么默认值就是空字符串。

配置示例如下所示:

```
<fieldType name="text_pattern_replace" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.PatternReplaceCharFilterFactory"
  pattern="(aa)\d+(bb)" replacement="$1 $2" />
</analyzer>
</fieldType>
```

### (16) NGramTokenFilter

NGramTokenFilter 根据给定的 GramRange 对 Token 生成 n-gram token。它的工厂类是 NGramFilterFactory。NGramFilterFactory 有两个可选参数:

- 1) minGramSize: 最小 n-gram 大小, 必须大于 0, 默认值是 1;
- 2) maxGramSize: 最大 n-gram 大小, 必须大于 0, 默认值是 2。且必须 maxGramSize >= minGramSize。

配置示例如下所示:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="6"/>
</analyzer>
```

### (17) EdgeNGramTokenFilter

跟 NGramTokenFilter 类似, 不同的是 EdgeNGramTokenFilter 每次都是从最左边开始 n-gram。它的工厂类是 EdgeNGramFilterFactory。

EdgeNGramFilterFactory 有两个可选参数:

- 1) minGramSize: 最小 n-gram 大小, 必须大于 0, 默认值是 1;
- 2) maxGramSize: 最大 n-gram 大小, 必须大于 0, 默认值是 1。

配置示例如下所示:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.EdgeNGramFilterFactory" minGramSize="2" maxGramSize="4"/>
</analyzer>
```

### (18) ClassicFilter

ClassicFilter 分割 Token 的行为与 ClassicTokenizer 很相似，具体请看 ClassicTokenizer 的使用说明。ClassicFilter 的工厂类是 ClassicFilterFactory，没有可选的配置参数。

配置示例如下所示：

```
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory" />
<filter class="solr.ClassicFilterFactory" />
</analyzer>
```

### (19) CommonGramsFilter

CommonGramsFilter 会将字典文件里出现的 Token 与它后面的一个 Token 使用下划线 \_ 拼接起来组成一个新的 Token，这个新的 Token 作为前一个 Token 的同义词，并且新 Token 的 type=gram。比如“boy friend”会生成一个“boy\_friend”并且它是 boy 的同义词，这样用户搜索“boy”就能搜到包含 boyfriend 的文档。

它的工厂类是 CommonGramsFilterFactory，拥有的可选配置参数如下：

- ❑ words：字典文件的加载路径，可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录；
- ❑ format：如果字典文件是符合 snowball 格式的，那么 format=snowball，否则 format 可以不指定或者指定任意值，当 format 参数不指定，内部会默认去加载 core/conf 目录下的 stopwords.txt 停用词字典文件；
- ❑ ignoreCase：匹配字典文件里的字符时是否忽略大小写。默认不忽略。

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.CommonGramsFilterFactory" words="common_grams_words.txt"
ignoreCase="true"/>
</analyzer>
```

common\_grams\_words.txt 字典文件配置示例如下所示：

```
ex
boy
```

比如：

输入：ex boy friend

输出：ex ex-boy boy boy-friend friend

### (20) NumericPayloadTokenFilter

NumericPayloadTokenFilter 用于给指定的类型的 Token 添加一个 float 数字的 payload 属性。关于 token type 详情请查阅 org.apache.lucene.analysis.Tokenclass 类的源码。

NumericPayloadTokenFilter 的工厂类是 NumericPayloadTokenFilterFactory，拥有如下可选参数：

payload：必需参数，float 类型的数字，会被添加到匹配到的所有 token 的 payload 属性上。

typeMatch：必需参数，token type 的名称，只有类型匹配的 token 才会被添加 payload 属性。

配置示例如下所示：

```
<fieldType name="text_numeric_payload" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.NumericPayloadTokenFilterFactory"
payload="0.75" typeMatch="word"/>
</analyzer>
</fieldType>
```

比如：

输入：big bang boom boom boom

输出：big[0.75] bang[0.75] boom[0.75] boom[0.75] boom[0.75]

### (21) RemoveDuplicatesTokenFilter

RemoveDuplicatesTokenFilter 用于移除重复的 Token，如果两个 Token 的文本内容相同且 position 位置信息也相同那么就判定为重复 Token。RemoveDuplicatesTokenFilter 的一个使用场景就是比如有两个同义词“TV”，“TVs”，经过 EnglishMinimalStemFilter 处理后，“TVs”也变成了“TV”，此时就出现了两个“TV”同义词 Token，需要移除一个避免重复。

它的工厂类是 RemoveDuplicatesTokenFilterFactory，配置示例如下所示：

```
<analyzer>
<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"/>
<filter class="solr.EnglishMinimalStemFilterFactory"/>
<filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
```

### (22) SuggestStopFilter

SuggestStopFilter 跟 StopFilter 很相似，它也会丢弃在停用词字典文件里出现的 Token，与 StopFilter 不同的是，它不会移除最后一个停用词的 Token，除非它后面跟随的是一个 Token 分隔符（比如 -\_+=!@#%&\*)({}[]\/?><., 等）。保留下来的停用词会被添加上 Keyword 属性标记。它的工厂类是 SuggestStopFilterFactory，拥有的可选配置参数如下：

□ words：指定停用词字典文件的加载路径，可选参数。默认值是 StopAnalyzer# ENGLISH\_STOP\_WORDS\_SET；

□ format：定义停用词字典文件如何被解析，如果 words 参数未指定，那么 format 参数也不能指定。它的可选值有两种：

1) wordset：wordset 是默认格式，它支持每行一个单词，空白行和以 # 号开头的注释行会被忽略；

2) snowball：这种允许在一行指定多个单词，空白行会被忽略，使用垂直线 | 进行注释。

□ ignoreCase：匹配停用词时是否忽略大小写，默认值 false 即不忽略。

配置示例如下所示：

```
<analyzer type="query">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SuggestStopFilterFactory" ignoreCase="true"
    words="stopwords.txt" format="wordset"/>
</analyzer>
```

比如：

输入："The Cat the #"

输出："cast", "}"

输入："The The"

输出："the"

### (23) TokenOffsetPayloadFilter

TokenOffsetPayloadFilter 会将 Token 的位置偏移量信息添加到 Token 的 payload 属性上。

它的工厂类是 TokenOffsetPayloadTokenFilterFactory。

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.TokenOffsetPayloadTokenFilterFactory"/>
</analyzer>
```

比如：

输入："big bang boom boom boom"

输出："big"[0,3], "bang"[4,8], "boom"[9,13], "boom"[14,18], "boom"[19,23]

### (24) TypeAsPayloadTokenFilter

TypeAsPayloadTokenFilter 用将 Token 的 type 类型添加到 Token 的 payload 属性上。它的工厂类是 TypeAsPayloadTokenFilterFactory，没有可选的配置参数。

配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.TypeAsPayloadTokenFilterFactory"/>
</analyzer>
```

### (25) DelimitedPayloadTokenFilter

`DelimitedPayloadTokenFilter` 用于从 `Token` 的原始文本中提取出 `payload` 信息并添加到自身的 `payload` 属性上。`Token` 的原始文本中需要包含一个分隔符 `delimiter`，分隔符之前的被认定为 `Token` 的真实文本，而分隔符之后的被认定为 `payload` 信息。举个例子，假如分隔符是垂直线“|”，那么“foo|bar”中 `foo` 就是 `token`，而 `bar` 就是 `payload`。你可以指定一个 `PayloadEncoder` 编码器用于编码 `payload`。



**注意** 你必须确保你的 `Tokenizer` 不会剔除掉分隔符，否则 `DelimitedPayloadTokenFilter` 无法正常工作。

`DelimitedPayloadTokenFilter` 的工厂类是 `DelimitedPayloadTokenFilterFactory`，拥有如下可选的配置参数：

□ **encoder**：用于指定 `payload` 编码器，必需参数。`payload` 编码器对分隔符后面的文本进行 `payload` 编码，因为 `payload` 信息底层都是使用 `byte` 字节数组存储的，所以需要将数字或者字符串编码成字节数组。`encode` 参数可选值有：

1) **identity**：将字符数组转换成字节数组的编码器；

2) **integer**：将 `Integer` 数字转换成字节数组的编码器；

3) **float**：将 `float` 类型的数字转换成字节数组的编码器；

4) 如果你指定的 `encode` 值不在上述 3 个可选值范围内，那么你提供的 `encoder` 参数值必须是实现了 `PayloadEncoder` 接口的自定义类的完整包路径，如果你提供的类路径不正确会导致自定义编码器类加载失败从而抛出异常。

□ **delimiter**：`token` 与 `payload` 信息的分隔符，可选参数。默认值是垂直线“|”

配置示例如下所示：

```
<fieldType name="text_delimited_payload" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.DelimitedPayloadTokenFilterFactory"
      encoder="float" delimiter="|" />
  </analyzer>
</fieldType>
```

比如：



输入: big|0.7 bang|0.2 boom boom|1.6 boom

输出: big[0.7] bang[0.2] boom boom[1.6] boom

### (26) PatternCaptureGroupTokenFilter

PatternCaptureGroupTokenFilter 会根据指定的正则表达式的捕获组从原始的 Token 中提取出 N 个新的 Token, 每个捕获组匹配到的文本对应一个 Token, 而生成的这些新 Token 会作为原始 Token 的同义词, 即它们的 positionIncrement=0。

PatternCaptureGroupTokenFilter 的工厂类是 PatternCaptureGroupTokenFilterFactory, 拥有如下可选的配置参数:

- ☐ pattern: Java 里的正则表达式, 必需参数;
- ☐ preserve\_original: 是否保留原始 Token, 可选参数, 默认值是 true, 即会保留原始 Token。

配置示例如下所示:

```
<fieldType name="text_pattern_capture_group" class="solr.TextField">
<analyzer>
<tokenizer class="solr.WhitespaceTokenizerFactory"/>
<filter class="solr.PatternCaptureGroupTokenFilterFactory"
    pattern="([A-Za-z]{3})" preserve_original="true" />
</analyzer>
</fieldType>
```

比如:

输入: abcdefghiJKLMNOP

输出: abcdefghiJKLMNOP abc def ghi JKL MNO

### (27) DictionaryCompoundWordTokenFilter

DictionaryCompoundWordTokenFilter 用于对复合词进行断词, 复合词指的是类似 handbag、boyfriend、goodnight 这类词, 你可以希望分成 hand 和 bag、boy 和 friend、good 和 night, 这样用户搜索 hand 或 bag 都可以搜出 handbag, boyfriend 同理。对于 handbag 这类复合词其实就是骆驼命名法的文本, 可以使用 WordDelimiterFilter 来处理。DictionaryCompoundWordTokenFilter 是基于字典文件来拆分复合词的, 分出的新 Token 会作为原始 Token 的同义词。

DictionaryCompoundWordTokenFilter 的工厂类是 DictionaryCompoundWordTokenFilterFactory, 拥有如下可选的配置参数:

- ☐ dictionary: 定义复合词字典文件加载路径, 既可以是绝对路径也可以是相对路径, 相对路径是相对 core/conf 目录, 必需参数;
- ☐ minWordSize: 表示原始 Token 的字符最小长度, 若小于此长度则不做断词处理, 默认值 5;
- ☐ minSubwordSize: 复合词断词后分出的子 Token 的最小长度, 若小于此长度则忽略,

默认值 2;

□ `maxSubwordSize` : 复合词断词后分出的子 Token 的最大长度, 若大于此长度则忽略, 默认值 15;

□ `onlyLongestMatch` : 复合词断词后分出 N 个子 Token, 是否只取长度最大的那个子 Token, 其他子 Token 都忽略, 默认为 true。

配置示例如下所示:

```
<fieldType name="text_dictionary_compound_word" class="solr.TextField">
<analyzer>
<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.WordDelimiterFilterFactory"
generateWordParts="1" generateNumberParts="0" catenateWords="0"
catenateNumbers="0" catenateAll="0" splitOnCaseChange="1"/>
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.DictionaryCompoundWordTokenFilterFactory"
dictionary="compound-words.txt" minWordSize="3" minSubwordSize="2"
maxSubwordSize="15" onlyLongestMatch="false"/>
<filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt" />
<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
<filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
</fieldType>
```


复合词字典文件 `compound-words.txt` 配置示例如下所示:

```
hand
bag
boy
friend
```

比如:

输入: boyfriend

输出: boyfriendboyfriend

 **注意** `DictionaryCompoundWordTokenFilter` 的字典文件里的单词可以任意组合, 比如 `handfriend`、`boybag` 也可以被断词, 虽然 `handfriend`、`boybag` 这类单词并不真实存在, 但总有可能你字典里的任意两个单词合并在一起会组合成一个新的单词, 而假如此时你不希望该新词被断开, 那就不一定有办法解决了。这也是 `DictionaryCompoundWordTokenFilter` 隐含一个缺点, 望知晓!

## (28) HyphenationCompoundWordTokenFilter

`HyphenationCompoundWordTokenFilter` 与 `DictionaryCompoundWordTokenFilter` 类似, 都是

用于对复合词进行断词，比如：boyfriend 可以分成 boy 和 friend。HyphenationCompoundWordTokenFilter 是 DictionaryCompoundWordTokenFilter 的功能增强版，HyphenationCompoundWordTokenFilter 支持与 DictionaryCompoundWordTokenFilter 同样的根据字典文件实现复合词断词，还支持根据 hyphenator.xml 规则文件进行断词。

HyphenationCompoundWordTokenFilter 的工厂类是 HyphenationCompoundWordTokenFilterFactory，拥有如下可选的配置参数：

- ❑ hyphenator：定义断词规则文件的加载路径，既可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录，必需参数；
- ❑ dictionary：定义字典文件的加载路径，既可以是绝对路径也可以是相对路径，相对路径是相对 core/conf 目录，可选参数；
- ❑ minWordSize：表示原始 Token 的字符的最小长度，若小于此长度则不做断词处理，默认值 5；
- ❑ minSubwordSize：复合词断词后分出的子 Token 的最小长度，若小于此长度则忽略，默认值 2；
- ❑ maxSubwordSize：复合词断词后分出的子 Token 的最大长度，若大于此长度则忽略，默认值 15；
- ❑ onlyLongestMatch：复合词断词后分出 N 个子 Token，是否只取长度最大的那个子 Token，其他子 Token 都忽略，默认为 true；
- ❑ encoding：定义断词规则文件的编码，可选参数，默认值是 UTF-8。

配置示例如下所示：

```
<fieldType name="text_hyphenation_compound_word" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.HyphenationCompoundWordTokenFilterFactory"
      hyphenator="hyphenator.xml"
      encoding="UTF-8" minWordSize="2" minSubwordSize="2"
      maxSubwordSize="15" onlyLongestMatch="false">
    </filter>
  </analyzer>
</fieldType>
```

断词规则文件 hyphenator.xml 配置示例如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hyphenation-info SYSTEM "hyphenation.dtd">
<!--
This file contains the hyphenation patterns for danish.
Adapted from dkhyph.tex, dkcommon.tex and dkspecial.tex
originally created by Frank Jensen (fj@iesd.auc.dk).
FOP adaptation by Carlos Villegas (cav@uniscopes.co.jp)
-->
```

```
<hyphenation-info>
<hyphen-char value="-"/>
<hyphen-min before="2" after="2"/>
```

```
<classes>
```

```
aA
```

```
bB
```

```
cC
```

```
dD
```

```
eE
```

```
fF
```

```
gG
```

```
hH
```

```
iI
```

```
jJ
```

```
kk
```

```
lL
```

```
mM
```

```
nN
```

```
oO
```

```
pP
```

```
qQ
```

```
rR
```

```
sS
```

```
tT
```

```
uU
```

```
vV
```

```
wW
```

```
xX
```

```
yY
```

```
zZ
```

```
æÆ
```

```
øØ
```

```
åÅ
```

```
</classes>
```

```
<patterns>
```

```
.girl5friend
```

```
.boy5friend
```

```
.hand5bag
```

```
.mother5day
```

```
.book5shop
```

```
.home5work
```

```
.T5Shirt
```

```
.butter5fly
```

```
</patterns>
```

```
</hyphenation-info>
```

规则定义在 `<patterns>``</patterns>` 之间, 比如 `.girl5friend` 开头的点号表示单词的边界, `girl` 与 `friend` 之间的数字 5 表示在数字 5 占据的位置处断开的可能性级别, 可能性级别用数字表示, 级别分 5 个等级, 取值范围 [1,5], 数字越大, 断开的可能性越大。你可以定义

多个断开点，比如 `.g1irl2friend`，表示在字母 `g` 后面断开的可能性级别是 1，在小写字母 `l` 后面断开的可能性是 2，按照理论此时应该在小写字母 `l` 后面断开，分成 `girl` 和 `friend`。但 `hyphenator` 规则里还有一条：最终选择的断开点的可能性级别数字必须是奇数，所以此时应该在可能级别为 1 的地方断开，分成 `g` 和 `irlfriend`。但是如果规则定义成这样：`g1irl5friend`，那么此时就应该分别在数字 1 和数字 5 处断开，分成 `g`、`irl` 和 `friend`。因为 1 和 5 都是奇数。

`<classes>` 与 `</classes>` 元素之间定义的是字符等价映射，上面示例里定义的字符映射含义就是忽略大小写的。

### (29) CachingTokenFilter

`CachingTokenFilter` 用于缓存其他 `TokenFilter` 生成的 `tokenStream` 属性，当一个 `TokenFilter` 被多次调用执行时，为避免对同一个 `TokenFilter` 重复计算生成 `tokenStream` 属性，所以当使用 `CachingTokenFilter` 的第一次调用时，就将生成的 `tokenStream` 属性缓存到一个 `List` 集合中，后续重复执行同一个 `TokenFilter` 时直接从缓存中获取。

Solr 并没有为 `CachingTokenFilter` 设计 `CachingTokenFilterFactory`，而在 Solr 中要使用 `TokenFilter`，必须配置对应的 `TokenFilterFactory`，`TokenFilter` 并不能直接使用。所以我们需要自己实现 `CachingTokenFilterFactory`。自定义 `TokenFilterFactory` 需要继承 `TokenFilterFactory` 基类，重写它的 `public TokenFilter create (TokenStream input)` 函数，在 `create` 函数内部创建对应的 `TokenFilter` 实现类。如果需要自定义外部参数，你需要在自定义的 `TokenFilterFactory` 的构造函数内部通过 `get` 函数获取外部自定义参数。自定义外部参数示例如下：

```
String dicFile = get(args, "dictionary"); // 获取 String 类型参数
String hypFile = require(args, "hyphenator"); // require 用于获取必需传递的参数
int minWordSize = getInt(args, "minWordSize" 2); // 获取 int 类型参数，第 3 个参数表示默认值。第 2 个参数表示自定义的外部参数的名称
```

同理还有 `getFloat()`、`getBoolean()` 等。这样你就可以在 `schema.xml` 的 `<filter>` 元素里添加自定义的外部参数。配置示例如下所示：

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="xx.xxxxxx.CachingTokenFilterFactory"/>
</analyzer>
```

## 4.2.4 CharFilter

`CharFilter` 是一个对输入文本进行预处理的组件。它也可以像 `Tokenizer` 和 `TokenFilter` 那样加入分词处理链条中，它需要配置在 `Tokenizer` 之前。`CharFilter` 可以添加、更新、删除字符，但会保留原始文本的位置偏移量，从而保证对高亮功能的支持。

`CharFilter` 是 `java.io.Reader` 的子类，而 `Tokenizer` 的构造函数刚好可以接收一个 `java`。

io.Reader，因此 CharFilter 可以作为 Tokenizer 的预处理，体现到 schema.xml 里就是，你可以在 <tokenizer> 元素之前配置 N 个 <charFilter> 元素，这 N 个 <charFilter> 元素就形成了 CharFilter 字符过滤器链条。

(1) MappingCharFilter

MappingCharFilter 用于将一个字符映射成另一个字符，比如 é → e。

它的工厂类是 MappingCharFilterFactory，拥有的配置参数如下所示：

mapping：字符映射文件的加载路径，既可以是绝对路径，也可以是相对路径，相对路径是相对 core/conf 目录下。

配置示例如下所示：

```
<analyzer>
  <charFilter class="solr.MappingCharFilterFactory" mapping="mapping-FoldToASCII.txt"/>
  <tokenizer ...>
    [...]
  </tokenizer>
</analyzer>
```

字符映射文件的编写语法：

- 以 # 号开头的行以及空白行将会被忽略；
  - 每个非注释行非空格白的映射格式：“source”=>“target”；
  - source 表示源字符，target 表示映射后的目标字符，source 和 target 都需要使用双引号包裹起来；
  - 在映射行的末尾添加 # 号是不被允许的；
  - source 必须至少包含一个字符，但 target 可以为空字符。
- source 和 target 都支持的特殊转义字符如表 4-1 所示。

表 4-1 特殊转义字符表

转义字符	含义	示例
\\	\	"\\" => "/"
\"	"	"\"and\"" => "and"
\b	对应键盘上的 Backspace 键。	"\b" => " "
\t	对应键盘上的 Tab 键。	"\t" => ", "
\n	换行符，将当前位置移到下一行开头。	"\n" => " "
\f	换页符，将当前位置移到下页开头。	"\f" => "\n"
\r	回车符，将当前位置移到本行开头。	"\r" => "/carriage-return/"
\uXXXX	4 个十六进制数字表示的 Unicode 字符。	"\uFEFF" => ""

(2) HTMLStripCharFilter

HTMLStripCharFilter 用于对 HTML 进行过滤，然后将处理结果传递给另一个 CharFilter 或者 Tokenizer。



HTMLStripCharFilter 支持如下功能：

- ❑ 移除 HTML/XML 的标签，只保留文本内容；
- ❑ 移除标签的属性并支持可选属性引用；
- ❑ 移除 XML 处理指令；
- ❑ 移除 XML 注释；
- ❑ 移除 XML 中以 `<!--` 开头的元素；
- ❑ 移除 `<script>` 和 `<style>` 元素；
- ❑ 处理在元素内部的 XML 注释（通常注释处理并不保证始终好使）；
- ❑ 替换数字字符实体引用（比如 `&#65` 或 `&#x7f`）成相应的字符；
- ❑ 如果实体引用在输入的末尾，那么结束符 `;"` 是可选的，为了避免像 "Alpha&Omega Corp" 这样的字符匹配失败，结束符 `;"` 是强制的；
- ❑ 替换所有命名字符实体引用成相应的字符，比如使用 `&nbsp` 替换 `0xa0` 字符；
- ❑ 换行使用块级元素替代；
- ❑ 支持对 `<CDATA>` 部分的识别；
- ❑ 行内标签，如 `<b>`、`<i>` 或 `<span>` 将会被移除；
- ❑ 支持对大写字符实体比如 `quot`、`gt`、`lt` 以及 `amp` 的识别并转换成小写形式。

下面的表格演示了 HTMLStripCharFilter 的一些示例：

表 4-2 HTMLStripCharFilter 示例

输 入	输出
my <a href="www.foo.bar">link</a>	my link
 hello<!--comment-->	hello
hello<script><!-- f(<!--internal--></script>); --></script>	hello
if a<b then print a;	if a<b then print a;
hello <td height=22 nowrap align="left">	hello
a<b &#65 Alpha&Omega Ω	a<b A Alpha&Omega

它的工厂类是 HTMLStripCharFilterFactory，没有可选的配置参数。

配置示例如下所示：

```
<analyzer>
  <charFilter class="solr.HTMLStripCharFilterFactory"/>
  <tokenizer ...>
  [...]
</analyzer>
```

### （3）ICUNormalizer2CharFilter

ICUNormalizer2CharFilter 会借助 ICU4J 对输入文本进行 Unicode 标准化预处理操作。

它的工厂类是 ICUNormalizer2CharFilterFactory，拥有以下可选的配置参数：

- ❑ name: 表示一个 Unicode 标准化格式, 可选值有 nfc, nfkc, nfkc\_cf。默认值是 nfkc\_cf。
- ❑ mode: 可选值: compose/decompose, 默认值是 compose。
- ❑ filter: 表示一个 UnicodeSet 的表达式, 默认值是 []。
- ❑ UnicodeSet 表达式配置示例如表 4-3 所示。

表 4-3 Unicode Set 表达式配置示例

表达式	描述
[]	不过滤任何字符
[a]	过滤字符 a
[ae]	过滤字符 a 和 e
[a-e]	过滤字符 a-e
[\u4E01]	过滤字符 U+4E01
[a{ab}{ac}]	过滤字符 "ab" 和 "ac"

配置示例如下所示:

```
<analyzer>
  <charFilter class="solr.ICUNormalizer2CharFilterFactory"/>
  <tokenizer ...>
    [...]
  </tokenizer>
</analyzer>
```

#### (4) PatternReplaceCharFilter

PatternReplaceCharFilter 用于使用 Java 里的正则表达式来替换或更新字符。它的工厂类是 PatternReplaceCharFilterFactory, 拥有如下可选的配置参数:

- ❑ pattern: 表示 Java 里的正则表达式。
- ❑ replacement: 表示替换字符。

配置示例如下所示:

```
<analyzer>
  <charFilter class="solr.PatternReplaceCharFilterFactory"
    pattern="([nN][oO]\.)*s*(\d+)" replacement="$1$2"/>
  <tokenizer ...>
    [...]
  </tokenizer>
</analyzer>
```

表 4-4 演示了一些正则表达式替换示例:

表 4-4 正则表达式替换示例

输入	正则表达式	替换字符	输出	描述
see-ing looking	(\w+)(ing)	\$1	see-ing look	移除末尾的 ing
see-ing looking	(\w+)ing	\$1	see-ing look	同上
No.1 NO. no. 543	[nN][oO]\.s*(\d+)	#\$1	#1 NO. #543	字符替换
abc=1234=5678	(\w+)=(\d+)=(\d+)	\$3=\$1=\$2	5678=abc=1234	改变字符顺序

## 4.2.5 Solr 自定义分词

我们知道，分词器主要是由 `Tokenizer` 和 `TokenFilter` 组成的，所以想要自定义分词器，首先需要自定义 `Tokenizer`，而 `TokenFilter` 是可选的，可以不需要自定义。当 Solr 内置的 `Tokenizer` 或 `TokenFilter` 已经无法满足需求时，你就可以考虑通过自定义的方式来进行功能扩展。

扩展 `Tokenizer`，你只需要继承 `Tokenizer` 抽象类（字符串级别的分词）或 `CharTokenizer` 抽象类（字符级别的分词），扩展 `TokenFilter`，你只需要继承 `TokenFilter` 抽象类。扩展 `Analyzer`，你需要继承 `SolrAnalyzer` 抽象类或 `Analyzer` 抽象类。

### 1. 自定义 Tokenizer

`Tokenizer` 的执行任务就是将输入文本分解成一个个 `Token`。假如有这样格式的字符串：`This+is+my+duty`。字符串的每个单词都是使用加号连接在一起的，我们可能希望使用加号作为分割符将它分成“`This`”“`is`”“`my`”“`duty`”。此时，我们就可以尝试自定义 `Tokenizer` 来实现。

首先你需要继承 `Tokenizer` 抽象类，重写它的 `incrementToken` 方法，示例代码如下所示：

```
import org.apache.lucene.analysis.Tokenizer;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;
import org.apache.lucene.util.AttributeFactory;
import java.io.IOException;
import java.io.Reader;
import static com.yida.book.solr5.demo.util.BaseTools.isBlankStr;
/**
 * Created by Lanxiaowei
 */
public class PlusSignTokenizer extends Tokenizer {
    /*Reader 流里的原始文本内容 */
    protected String stringToTokenize;
    /**
     * 当前位置
     */
    protected int position = 0;
    /**
     * 用于缓存 Token 的文本
     */
    protected CharTermAttribute charTermAttribute =
        addAttribute(CharTermAttribute.class);
    @Override
    public boolean incrementToken() throws IOException {
        // 先清空 charTermAttribute 缓存的数据
        this.charTermAttribute.setEmpty();
        // 若 stringToTokenize 未初始化，才初始化，后续不重复初始化
        if (isBlankStr(this.stringToTokenize)) {
```

```

        this.stringToTokenize = transformReader(input);
    }

    // 获取下一个加号的位置
    int nextIndex = this.stringToTokenize.indexOf('+', this.position);
    if (nextIndex != -1) {
        String nextToken = this.stringToTokenize.substring(
            this.position, nextIndex);
        // 提取出 Token 并缓存到
        this.charTermAttribute.append(nextToken);
        this.position = nextIndex + 1;
        return true;
    }

    // 处理最后一个加号后面剩余的文本
    else if (this.position < this.stringToTokenize.length()) {
        String nextToken =
            this.stringToTokenize.substring(this.position);
        this.charTermAttribute.append(nextToken);
        this.position = this.stringToTokenize.length();
        return true;
    }

    // 到此说明所有文本都处理完了
    else {
        return false;
    }
}

/**
 * 读取 Reader 里的数据并缓存到 stringToTokenize
 */
public PlusSignTokenizer(AttributeFactory factory) {
    super(factory);
}

/**
 * 重置 position 信息
 * @throws IOException
 */
@Override
public void reset() throws IOException {
    super.reset();
    this.position = 0;
}

public String transformReader(Reader reader) {
    int numChars;
    char[] buffer = new char[1024];
    StringBuilder stringBuilder = new StringBuilder();
    try {
        while ((numChars =
            reader.read(buffer, 0, buffer.length)) != -1) {
            stringBuilder.append(buffer, 0, numChars);
        }
    }

```

```

    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    this.stringToTokenize = stringBuilder.toString();
    return this.stringToTokenize;
}
}

```

接下来，我们可以编写代码对 `PlusSignTokenizer` 进行测试：

```

/**
 * Created by Lanxiaowei
 */
public class TestPlusSignTokenizer {
    public static void main(String[] args) throws Exception {
        String text = "Right+here+with+you";
        TokenStream stream = new PlusSignTokenizer(
            BaseTools.newAttributeFactory());
        ((Tokenizer)stream).setReader(new StringReader(text));
        String result = BaseTools.tokenstream2String(stream);
        System.out.println(result);
    }
}

```

代码中涉及的工具类代码如下所示：

```

/**
 * Created by Lanxiaowei
 * 基础工具类
 */
public class BaseTools {
    /**
     * 随机创建 AttributeFactory 工厂类
     * @return
     */
    public static AttributeFactory newAttributeFactory() {
        return TokenStream.DEFAULT_TOKEN_ATTRIBUTE_FACTORY;
    }

    /**
     * 创建 Random 对象
     * @return
     */
    public static Random random() {
        return new Random();
    }

    /**
     * 返回 SegmentReader
     */
}

```

[illegible]



```

    }

    /**
     * 字符转义
     *
     * @param s
     * @return
     */
    public static String escape(String s) {
        int charUpto = 0;
        final StringBuilder sb = new StringBuilder();
        while (charUpto < s.length()) {
            final int c = s.charAt(charUpto);
            if (c == 0xa) {
                // Strangely, you cannot put \ u000A into Java
                // sources (not in a comment nor a string
                // constant)...:
                sb.append("\\n");
            } else if (c == 0xd) {
                // ... nor \ u000D:
                sb.append("\\r");
            } else if (c == '"') {
                sb.append("\\\"");
            } else if (c == '\\') {
                sb.append("\\\\");
            } else if (c >= 0x20 && c < 0x80) {
                sb.append((char) c);
            } else {
                sb.append(String.format(Locale.ROOT, "\\u%04x", c));
            }
            charUpto++;
        }
        return sb.toString();
    }

    /**
     * 判断是否为空字符串或 Null
     *
     * @param str
     * @return
     */
    public static boolean isBlankStr(String str) {
        return null == str || str.length() == 0;
    }
}

```

最后代码测试输出结果为：Right here with you。与我们期望的一致。

## 2. 自定义 TokenFilter

上一章节中，我们只是按照加号 + 字符对字符串进行分割，没有考虑空格情况，所以

分出来的 Token 中有可能有空格，因此我们还需要自定义 TokenFilter 对 PlusSignTokenizer 进行过滤。

首先需要继承 TokenFilter 抽象类，重写其相应的方法。在 TokenFilter 类内部定义了几个可以重写的方法，如下所示：

```
/**
 * 当最后一个 Token 被消费完后，
 * 用户可以调用此方法做些额外操作
 */
@Override
public void end() throws IOException {
    input.end();
}

/**
 * 当最后所有 Token 都消费完后，
 * 用户可以调用此方法来释放当前 TokenFilter 占用的资源
 */
@Override
public void close() throws IOException {
    input.close();
}

/**
 * 当用户在调用 incrementToken 方法遍历 Token 之前，
 * 先调用此 reset 方法重置当前 TokenFilter 的状态
 */
@Override
public void reset() throws IOException {
    input.reset();
}
}

/**
 * 重写此方法实现遍历每个 Token 的代码逻辑
 * 每次通过调用此方法遍历下一个 Token，
 * 如果 return false，则表示已经遍历到底了。
 */
public abstract boolean incrementToken() throws IOException;
```

其中只有 incrementToken 方法是强制必须重写的，其他方法酌情选择。这里首先创建一个 EmptyStringTokenFilter 继承 TokenFilter，重写其 incrementToken 方法，示例代码如下所示：

```
import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;
import org.apache.lucene.analysis.tokenattributes.PositionIncrementAttribute;

import java.io.IOException;
```

```

/**
 * Created by Lanxiaowei
 */
public class EmptyStringTokenFilter extends TokenFilter {

    public EmptyStringTokenFilter(TokenStream tokenStream) {
        super(tokenStream);
    }

    // 用于缓存当前 Token 的文本
    protected CharTermAttribute charTermAttribute =
        addAttribute(CharTermAttribute.class);
    // 用于缓存当前 Token 的位置增量
    protected PositionIncrementAttribute positionIncrementAttribute =
        addAttribute(PositionIncrementAttribute.class);

    /**
     * 遍历每个 Token
     * @return
     * @throws IOException
     */
    @Override
    public boolean incrementToken() throws IOException {
        clearAttributes();
        String nextToken = null;
        while (nextToken == null) {

            // 遍历 tokenizer 传递过来的每个 Token, 如果已经遍历到底了
            if (! this.input.incrementToken()) {
                return false;
            }

            // 获取 tokenizer 缓存的当前 Token
            String currentTokenInStream =
                this.input.getAttribute(CharTermAttribute.class)
                    .toString().trim();

            // 如果不是一个空字符串, 则缓存到 nextToken
            if (null != currentTokenInStream &&
                currentTokenInStream.length() > 0) {
                nextToken = currentTokenInStream;
            }

            // 保存当前 Token
            this.charTermAttribute.setEmpty().append(nextToken);
            // 设置位置增量
            this.positionIncrementAttribute.setPositionIncrement(1);
            return true;
        }
    }
}

```

同理，我们需要遍历代码对其进行功能测试，示例如下所示：

```
public class TestEmptyStringTokenFilter {
    public static void main(String[] args) throws Exception {
        // 在字符串里增加空格进行测试
        String text = "Right+ here+with +you";
        TokenStream tokenizer = new PlusSignTokenizer(
            BaseTools.newAttributeFactory());
        ((Tokenizer)tokenizer).setReader(new StringReader(text));
        TokenStream tokenFilter = new EmptyStringTokenFilter(tokenizer);
        String result = BaseTools.tokenstream2String(tokenFilter);
        System.out.println(result);
    }
}
```

到此 Tokenizer 和 TokenFilter 都编写完毕了，暂时还只能通过 Lucene API 进行测试，如果想要将其配置到 Solr 的 Schema 中，我们还需要编写相应的 TokenizerFactory 和 TokenFilterFactory。无论是 TokenizerFactory 还是 TokenFilterFactory，都只需要继承基类并重写其 create 方法并创建对应的 Tokenizer 和 TokenFilter 即可，示例代码如下所示（仅供参考）：

#### (1) PlusSignTokenizerFactory 类

```
/**
 * Created by Lanxiaowei
 */
public class PlusSignTokenizerFactory extends TokenizerFactory {
    /// 这里的 args 是用于接收用户在 schema 的 <tokenizer> 元素里自定义的属性值
    protected PlusSignTokenizerFactory(Map<String, String> args) {
        super(args);
    }

    @Override
    public Tokenizer create(AttributeFactory attributeFactory) {
        return new PlusSignTokenizer(attributeFactory);
    }
}
```

#### (2) PlusSignTokenFilterFactory 类

```
/**
 * Created by Lanxiaowei
 */
public class EmptyStringTokenFilterFactory extends TokenFilterFactory {
    // 这里的 args 是用于接收用户在 schema 的 <filter> 元素里自定义的属性值
    protected EmptyStringTokenFilterFactory(Map<String, String> args) {
        super(args);
    }

    @Override
    public TokenStream create(TokenStream tokenStream) {
```

```

        return new EmptyStringTokenFilter(tokenStream);
    }
}

```

编写完毕后，你还必须将上述类打包成 jar，然后将 jar 包放置到 core/lib 目录下，然后 reload 你的 Core 使其重新加载依赖的 jar 包，从而能够立即生效。剩下就是在 schema.xml 里配置 Tokenizer 和 filter，至于怎么配置 Tokenizer 和 TokenFilter，前面章节已提供了大量示例，这里不再赘述，如若仍不清楚，请翻阅前面内容进行回顾。不过还是要提醒下，在配置自定义的 Tokenizer 和 TokenFilter 的实现类时，由于此时实现类不是 Solr 内部自带的，不能使用 solr. 前缀来引用，必须提供 Tokenizer 和 TokenFilter 的完整包路径哦。能够使用 solr. 前缀作为包名的前提是你的实现类的包名是 org.apache.solr.(search|update|request|core|analysis)。

### 3. 自定义 Analyzer

虽然在 Solr 里推崇你通过 <tokenizer>+<tokenFilter> 任意组合方式来实现自己复杂的个性化 fenci 需求，这本来也是一种很好的设计模式体现，即在设计之初，个体与个体之间应该尽量多分离少聚合，从而降低耦合度，最后交由用户去任意组合，增强灵活性。

而自定义 Analyzer 其实就是在类内部完成 Tokenizer 与 TokenFilter 的组合作，虽然 Solr 也允许你在 schema.xml 中直接配置 Analyzer，但这不是一种优雅的实现方式。不过这里还是需要对如何自定义 Analyzer 稍作说明，大家了解即可。因为自定义 Analyzer 大概只有在 Lucene 中才会用到。

首先你需要继承 Analyzer 抽象类，重写其 createComponents 方法，在其内部构造分词的处理链，具体实现代码如下所示：

```

/**
 * Created by Lanxiaowei
 */
public class PlusSignAnalyzer extends Analyzer {
    /**
     * 创建 TokenStreamComponent，在内存组合分词处理链条
     * @param fieldName 此参数表示域名称，即当前分词器应用到哪个域上
     * @return
     */
    protected TokenStreamComponents createComponents(String fieldName) {
        Tokenizer tokenizer = new
        PlusSignTokenizer(TokenStream.DEFAULT_TOKEN_ATTRIBUTE_FACTORY);
        TokenStream filter = new EmptyStringTokenFilter(tokenizer);
        // 添加一个转换小写的过滤器
        filter = new LowerCaseFilter(filter);
        return new TokenStreamComponents(tokenizer, filter);
    }
}

```

`createComponents` 方法是强制重写的, `Analyzer` 抽象类内部还提供了几个可选方法供用户去重写, 所以有必要了解提供的这些方法到底是做什么用途, 以及这些方法在什么时候被调用, 只有这样你才知道怎么去重写它。

比如 `initReader` 方法, 定义如下所示:

```
// fieldName: 域名称, reader: 域的域值 (只不过是内部会自动将其转换为 Reader 字符流类型)
protected Reader initReader(String fieldName, Reader reader)
```

`initReader` 方法表示初始化数据源, 当你希望对数据源进行一些预处理时, 你可以重写此方法。这里说的数据源就是需要分词的原始数据, 这里使用 `java` 的 `Reader` 字符流来表示原始数据, 之所以使用 `Reader` 字符流来统一表示待分词数据源, 是为了兼容各种类型数据源情况, 因为我们待分词的数据源可能来自外部文件内部的文本, 也有可能来自网络上的一个远程文件, 还有可能来自外部的数据库系统中, 最理想的情况就是用户直接传入一个 `String` 类型字符串。创建 `Field` 时可能是这样实现的:

```
new TextField(fieldName, "这里是待分词的文本字符串", Field.Store.YES);
```

我们输入的待分词的文本字符串就是数据源, 如果它是 `String` 类型, `Solr` 内部会自动将其转换成 `StringReader`。因此, 你可以将方法的第二个参数理解为你创建域时传递的域值。我们创建索引文档 `Document` 时, 可能是这样的:

```
indexwriter.addDocument(doc);
```

`indexwriter` 在创建索引之前, 首先需要对 `Document` 的每个 `Field` 的域值进行分词, 而分词之前会先调用 `initReader` 方法, 也就是说你可以在分词之前, 通过重写 `initReader` 方法改变用户传入的待分词的文本内容。举个例子, 比如用户传入的待分词文本是一串 `HTML` 格式的字符串, 里面包含了很多 `HTML` 标签, 类似 `<div>`、`<span>` 等元素, 而这些字符串通常是不需要进行索引的, 我们实际关心的可能是标签之间的文本, 此时你就可以尝试重写 `initReader` 方法, 在方法内部通过 `XPath` 或者 `XML` 解析类库去提取你感兴趣的内容, 最后返回新的 `Reader` 对象给 `TokenStreamComponents` 对象, `TokenStreamComponents` 随后会通过 `setReader` 方法将数据源 `Reader` 传递给自己内部维护的 `Tokenizer`, `Tokenizer` 抽象类内部的 `setReader(Reader input)` 方法就是用来接收外部的数据源, 而 `TokenStreamComponents` 的 `setReader` 方法其实就是在内部调用 `Tokenizer` 的 `setReader` 方法, 如下所示:

```
protected final Tokenizer source;
protected final TokenStream sink;
protected void setReader(final Reader reader) throws IOException {
    // 这里的 source 其实就是 Tokenizer 类型
    source.setReader(reader);
}
```

数据源初始化好了之后, `TokenStreamComponents` 会调用 `getTokenStream` 来返回 `TokenFilter`。因为我们在构造 `TokenFilter` 时, 为其传入了 `Tokenizer`, 对 `Tokenizer` 对象进行了装饰, 所

以实际分词时，是先从 `Tokenizer` 开始执行的，再执行多个 `TokenFilter` 链，依次执行各自的 `incrementToken` 方法遍历获取到每个 `Token`。

`Analyzer` 类内部还有一个可以重写的方法：`getPositionIncrementGap`，用于设置多值域的域值之间的间隙，而且此设置是针对应用了此 `Analyzer` 的多值域的多个域值之间间隙，不重写的话，它的默认值是 0。其实你可以通过 `setPositionIncrementGap()` 临时进行设置，如果你重写了之后只返回一个固定值，那就失去灵活性了。

如果你确实需要自定义 `Analyzer` 类并在 Solr 的 `schema.xml` 中配置以及应用它，那么你需要按照上述说明进行重写，代码编码完毕后将其打包成 jar 包，然后将 jar 包复制到 `core/lib` 目录下，然后 reload 你的 Core 使其立即生效。配置 `<analyzer>` 时通过为其添加 `class` 属性并指定属性值为自定义 `Analyzer` 类的完整包路径即可。配置示例如下所示：

```
<fieldType name="text_plus_sign" class="solr.TextField">
  <analyzer class="com.yida.book.solr5.demo.analyzer.PlusSignAnalyzer"/>
</fieldType>
```

在 Solr 中，自定义 `Analyzer` 除了继承 `Analyzer` 抽象类，还可以继承 `SolrAnalyzer` 抽象类，其实 `SolrAnalyzer` 就是 `Analyzer` 的一个子类，它内部什么都没做，只是将需要用户重写的方法再单独声明出来，提醒用户重写 `Analyzer` 其实只需要重写如下几个方法即可，也算是为用户排除了一些困扰吧，看起来更加简洁，一目了然：

```
public abstract class SolrAnalyzer extends Analyzer {
    int posIncGap = 0;
    public void setPositionIncrementGap(int gap) {
        posIncGap = gap;
    }
    @Override
    public int getPositionIncrementGap(String fieldName) {
        return posIncGap;
    }
    @Override
    protected Reader initReader(String fieldName, Reader reader) {
        return reader;
    }
}
```

至于是继承 Lucene 里的 `Analyzer` 抽象类呢，还是继承 Solr 里的 `SolrAnalyzer` 抽象类，我觉得是都可以的，唯一区别就是 `SolrAnalyzer` 抽象类是 Solr 提供的，即你需要额外依赖 Solr 的 jar 包，如果当前项目纯粹是基于 Lucene 的，没有使用到 Solr 而且你也不想额外的依赖 Solr，那么你就可以选择继承 Lucene 里的 `Analyzer` 抽象类，如果你项目中已经有了 Solr 环境，那么继承哪个请随便，代码重写起来没什么太大区别。不过还是建议选择继承 Lucene 里的 `Analyzer` 抽象类，因为如果选择继承 Solr 里的 `SolrAnalyzer` 抽象类，那么你的 `Analyzer` 类交给其他人使用时，会强制其他人添加 Solr 依赖，这不太友好，仅此而已。



## 4.3 中文分词器

所谓中文分词就是对中文语句中包含的词汇进行断词，分割出来的一个个包含独立含义的词语。当然，分出的词语最好能与原始语句的语义相吻合。中文分词一般同时还附带有去除标点符号，去除停用词，加载扩展字典添加对陌生词的分词支持，添加词性标注、语义标注。添加词性标注、语义标注，主要还是为了消除部分词语的歧义，比如：“你真的确实太棒啦”这样一句话，我们人类自然知道应该分成：你|真的|确实|棒。此时“的确”看起来也是一个词语，但在当前语境下，它不应该被分成一个词。因为这涉及根据上下文的语义环境来断词，也是分词领域的一个难题。

现有的分词算法有3大类：基于字典，基于理解，基于统计。

基于字典的分词方法就是扫描一个指定的汉语词典进行查找匹配，若找到了，则会分出一个词。根据扫描方式不同，又可以分为正向匹配和逆向匹配；根据不同长度优先匹配原则，又可以分为最长匹配和最短匹配。这几种匹配方式之间又可以互相组合。

基于理解的分词方式就是通过语义分析让计算机能够模拟人类理解句子的过程，从而达到分词目的。目前市面上还没有这类分词器。

基于统计的分词方式又分两种，一种是基于概率，即两个字相邻出现的次数越多，那么它们组成一个词的概率就越大。因此在分词阶段需要每个分词情况的相邻在一起的概率，最后返回概率值最大的作为最后分词结果。另一个就是基于统计学，就是利用统计模型来学习词语的切分规律，这个过程又称为训练，也就是说你需要用大量的语料库来模拟训练，从而最终实现分词。

对于英文来说，由于单词之间一般都是使用空格进行分割，有明显的分割界定符，而对于中文，由于中文中单字成词的情况较少，大部分都是双字词或多字词。而且中文在书写时，词语之间是粘连在一起的，没有明显的界定符。如果没有中文分词器，默认只能按照单字进行分词，这样会增大索引体积，影响查询效率不说，而且单字进行索引意味着返回的搜索结果相关性不高，用户体验极差。为了解决单字索引问题，有人提出对于中文进行二元分词，比如“我是中国人”会分成“我是”，“是中”，“中国”，“国人”。但这样依然会有查询相关性问题的，比如用户搜索“北大”会返回“东北大学”。分词是搜索的前提，没有好的分词效果，搜索体验也就无法保证，因此我们需要使用中文分词器来支持对中文进行分词。

Solr 中，默认只有一个 SmartCN 分词器，不过它被归入扩展包里，并没有纳入 Solr 的正式源码包里。而且 SmartCN 分词器的分词效果也不太理想。目前市面上的中文分词器常见的有：IK，MMSeg4J，Ansj，Paoding，Jcseg，ICTCLAS，FudanNLP。

### 4.3.1 IK 分词器

IKAnalyzer 是一个开源的，基于 java 语言开发的轻量级的中文分词工具包。IKAnalyzer 具有如下特性：

- 采用了特有的“正向迭代最细粒度切分算法”，支持细粒度和智能分词两种切分模式；
- 在系统环境：Core 2 i7 3.4G 双核，4G 内存，window 7 64 位，Sun JDK 1.6\_29 64 位普通 pc 环境测试，IK2012 具有 160 万字 / 秒（3000KB/S）的高速处理能力；
- 2012 版本的智能分词模式支持简单的分词排歧义处理和数量词合并输出；
- 采用了多子处理器分析模式，支持：英文字母、数字、中文词汇等分词处理，兼容韩文、日文字符；
- 优化的词典存储，更小的内存占用。支持用户词典扩展定义。特别的，在 2012 版本，词典支持中文，英文，数字混合词语。

目前，IK 分词器项目已经迁移托管到 OSChina 的 Git 仓库，访问地址如下所示：

<http://git.oschina.net/wltea/IK-Analyzer-2012FF>

遗憾的是，该项目的最后一次更新时间已经是 2 年前，目前最新版本并不支持 Solr 5.x，所以想要在 Solr 5.x 中使用 IK 分词器，需要自己修改源码。首先需要从 Git 仓库里下载一份 IK 当前的最新源码，然后导入到你的 IDEA 或 Eclipse 里。这里我直接给出修改后的源码，如下所示：

#### 1) IKTokenizer 类：

```
/**
 * IKTokenizer
 * 兼容 Lucene 5.x 版本
 */
public final class IKTokenizer extends Tokenizer {
    // IK 分词器实现
    private IKSegmenter _IKImplement;
    // 词元文本属性
    private final CharTermAttribute termAtt;
    // 词元位移属性
    private final OffsetAttribute offsetAtt;
    // 词元分类属性（该属性分类参考 org.wltea.analyzer.core.Lexeme 中的分类常量）
    private final TypeAttribute typeAtt;
    // 记录最后一个词元的结束位置
    private int endPosition;
    private Version version = Version.LATEST;
    public IKTokenizer(){
        // 默认细粒度切分算法
        this(false);
    }
    public IKTokenizer(boolean useSmart){
        offsetAtt = addAttribute(OffsetAttribute.class);
        termAtt = addAttribute(CharTermAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        _IKImplement = new IKSegmenter(input, useSmart);
    }
}
```

```

public IKTokenizer(AttributeFactory factory, boolean useSmart) {
    super(factory);
    offsetAtt = addAttribute(OffsetAttribute.class);
    termAtt = addAttribute(CharTermAttribute.class);
    typeAtt = addAttribute(TypeAttribute.class);
    _IKImplement = new IKSegmenter(input, useSmart);
}

@Override
public boolean incrementToken() throws IOException {
    // 清除所有的词元属性
    clearAttributes();
    Lexeme nextLexeme = _IKImplement.next();
    if(nextLexeme != null){
        // 将 Lexeme 转成 Attributes
        // 设置词元文本
        termAtt.append(nextLexeme.getLexemeText());
        // 设置词元长度
        termAtt.setLength(nextLexeme.getLength());
        // 设置词元位移
        offsetAtt.setOffset(nextLexeme.getBeginPosition(),
            nextLexeme.getEndPosition());
        // 记录分词的最后位置
        endPosition = nextLexeme.getEndPosition();
        // 记录词元分类
        typeAtt.setType(nextLexeme.getLexemeTypeString());
        // 返回 true 告知还有下个词元
        return true;
    }
    // 返回 false 告知词元输出完毕
    return false;
}

@Override
public void reset() throws IOException {
    super.reset();
    _IKImplement.reset(input);
}

@Override
public void end() throws IOException {
    super.end();
    int finalOffset = correctOffset(this.endPosition());
    offsetAtt.setOffset(finalOffset, finalOffset);
}
}

```

## 2) IKAnalyzer 类:

```

/**
 * IK 分词器 Lucene Analyzer 实现
 * 兼容 Lucene 5.x 版本

```

```

*/
public final class IKAnalyzer extends Analyzer{
    private boolean useSmart;
    public boolean useSmart() {
        return useSmart;
    }

    public void setUseSmart(boolean useSmart) {
        this.useSmart = useSmart;
    }

    /**
     * IK 分词器 Lucene Analyzer 接口实现类
     * 默认细粒度切分算法
     */
    public IKAnalyzer(){
        this(false);
    }

    /**
     * IK 分词器 Lucene Analyzer 接口实现类
     *
     * @param useSmart 当为 true 时，分词器进行智能切分
     */
    public IKAnalyzer(boolean useSmart){
        super();
        this.useSmart = useSmart;
    }

    /**
     * 重载 Analyzer 接口，构造分词组件
     */
    @Override
    protected TokenStreamComponents createComponents(String fieldName) {
        Tokenizer _IKTokenizer = new IKTokenizer(this.useSmart());
        return new TokenStreamComponents(_IKTokenizer);
    }
}

```

经过上述修改，IK 分词器就能在 Lucene 5.x 中正常使用了，使用的时候直接 new IKAnalyzer (useSmart)。这里的 useSmart 表示是否开启智能分词，如果设置为 false，那么就是按照细粒度进行切分。

如果想要在 Solr 中使用，我们还得自定义 IKTokenizerFactory，很简单，就直接继承 TokenFilterFactory，重写其 create (AttributeFactory attributeFactory) 方法，详细实现代码如下所示：

```

public class IKTokenizerFactory extends TokenizerFactory {
    public IKTokenizerFactory(Map<String, String> args) {
        super(args);
    }
}

```

```

        useSmart = getBoolean(args, "useSmart", false); // 默认细粒度切分
    }
    private boolean useSmart;

    @Override
    public Tokenizer create(AttributeFactory attributeFactory) {
        Tokenizer tokenizer = new IKTokenizer(attributeFactory, useSmart);
        return tokenizer;
    }
}

```

同理，你需要将其打包成 jar 然后放置到 core/lib 目录下，然后重新加载你的 Core。如果你需要在 Solr 5 中使用 IK 分词器，那么你还需把 IKAnalyzer.cfg.xml、ext.dic、stopword.dic 这 3 个文件 copy 到你的 Tomcat 的 webapps\solr\WEB-INF\classes 目录下。其中 IKAnalyzer.cfg.xml 配置如下所示：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>IK Analyzer 扩展配置</comment>
    <!-- 用户可以在这里配置自己的扩展字典，多个扩展字典用分号分割 -->
    <entry key="ext_dict">ext.dic;</entry>
    <!-- 用户可以在这里配置自己的扩展停止词字典 -->
    <entry key="ext_stopwords">stopword.dic;</entry>
</properties>


```

ext.dic 是 IK 分词器的扩展字典文件，用于添加一些 IK 分词器分不出来的词语，比如网络流行词“么么哒”，“狗带”，“高富帅”等。一个词语独占一行，且 ext.dic 字典文件的编码必须是 UTF-8 无 BOM 格式，否则会无法生效，因此，建议直接从 IK 源码包里复制一份再做修改，最好不要使用系统自带的记事本进行文字编辑，特此提醒！！

stopword.dic 是 IK 分词器的停用词字典文件，也是一个词语独占一行，文件的编码也必须是 UTF-8 无 BOM 格式。

扩展字典和停用词字典你可以拆分成多个文件分类存放，IKAnalyzer.cfg.xml 里支持配置多个字典文件路径，多个字典文件路径之间使用英文状态下的分号；进行分割。

---

 **注意** 不要敲成中文状态下的分号“；”，否则字典文件会加载失败，从而导致配置无法生效。

---

下面是 schema.xml 中使用 IK 分词器的配置示例：

```

<field name="name" type="text_ik" indexed="true" stored="true"
omitNorms="true"/>
<fieldType name="text_ik" class="solr.TextField">
<analyzer>
<tokenizerclass="org.apache.lucene.analysis.ik.IKTokenizerFactory"

```

```
useSmart="true"/>
</analyzer>
</fieldType>
```

但 IK 分词器对于阿拉伯数字与中文单位（比如天、千克、小时），阿拉伯数字 + 中文十进制位（比如百、千、万、百万、亿），阿拉伯数字 + 英文单位（比如：cm、km、kg、h、ml），中文数字 + 中文单位，中文数字 + 中文十进制位等等这类字符串组合，分词效果不太理想，为此，我修改了 IK 源码对其进行了改进。改动地方主要包括：添加了英文单位扩展字典文件，用于自动识别英文单位，并对 Token Type 做了明确区分，添加了连续数字的自动多次合并，默认源码内只会合并 2 次，添加了对中文数字和中文单位的自动识别，这样便于通过 Token Type 清楚数字与紧随其后的中文字符是否需要合并在一起。由于改动量太大，不便一一说明或贴出源码，有兴趣的同学请自行到我 Github 上去下载源码一探究竟（益达 Github 访问地址：<https://github.com/yida-lxw>）。这里给一张分词效果图你们感受下吧。

分词文本：T450 SKU-112 80KG 365 天八小时联想 2000 粤 TB01235 牛 B U 盘 G20 峰会分词后效果图如图 4-4 所示。

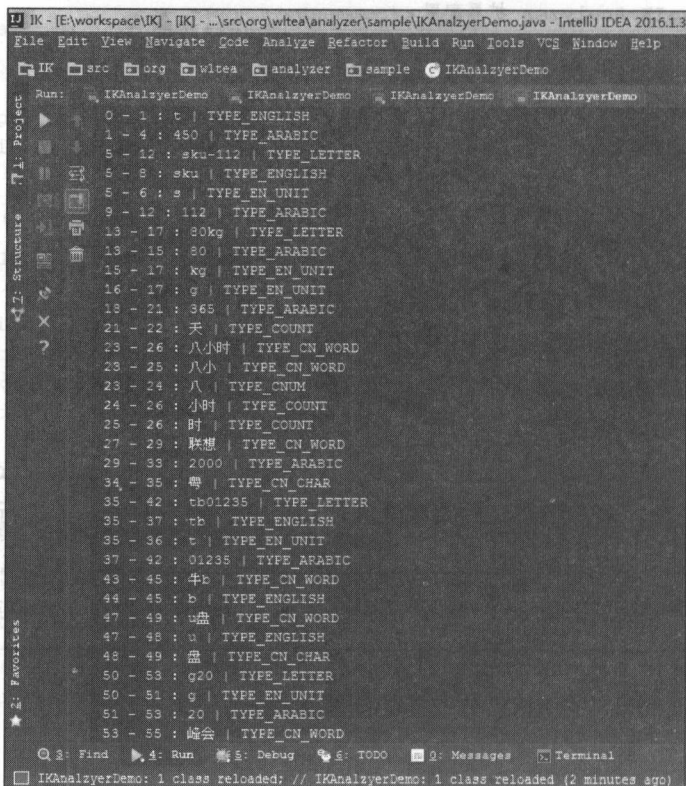


图 4-4 改进后的 IK 分词器的分词效果图

IK 分词器在歧义方面处理的还不够完善，有待改进。比如：当好人民的公仆，会分成：

当好 / 好人 / 人民 / 公仆。其中“好人”是不需要分出来的。

### 4.3.2 Ansj 分词器

Ansj 中文分词是一个完全开源的，基于 google 语义模型 + 条件随机场模型的中文分词的 Java 实现。它是 ictclas（汉语词法分析系统，由中科院采用采用 C/C++ 编写）的 Java 实现，基本上重写了所有的数据结构和算法。词典使用的是开源版的 ictclas 提供的，并且对其进行了部分人工优化。它具有以下特性：

- 使用简单开箱即用；
  - 内存中中文分词每秒钟大约 100 万字（速度上已经超越 ictclas），文件读取分词每秒钟大约 30 万字，准确率能达到 96% 以上；
  - 目前实现了中文分词、中文姓名识别、用户自定义词典、关键字提取、自动摘要、关键字标记等功能；
  - 支持词性标注；
  - 完全独立于 Lucene；
  - 可以应用到自然语言处理等方面，适用于对分词效果要求高的各种项目。
- 2016 年 7 月 30 日，Ansj 分词器 5.0.1 版本发布，主要做了如下几点更新：
- 修改了歧义词典的格式，修复 % 作为量词存在；
  - 对新词的发现提供了词性分析功能，词性不再全部标注为 nw；
  - 重新训练新词发现模型，针对机构名做了大量调优。

Ansj 分词器源码下载地址：[https://github.com/NLPchina/ansj\\_seg](https://github.com/NLPchina/ansj_seg)

在 Java 项目中使用 Ansj，你只需要访问 Ansj 的 github 地址，下载相关 jar 包，导入你的项目中。

Ansj 的 jar 包下载地址：<http://maven.nlpcn.org/org/ansj/>

如果你用的是 1.x 版本，你需要额外下载 tree\_split.jar；

如果你用的是 2.x 版本，你需要额外下载 nlp-lang.jar；

如果你用的是 3.x 以上版本，你需要额外下载 nlp-lang.jar，或者你直接下载 Ansj 作者提供的 ansj\_seg-[version]-all-in-one.jar 一个 jar 包就可以了。

至于 Ansj 各版本依赖的 tree\_split.jar 和 nlp-lang.jar 的具体版本，请下载 ansj\_seg-1.2.pom 文件进行查阅，里面清楚定义了当前版本的 Ansj 依赖哪些 jar 包，比如 Ansj 5.0.2 版本的 pom 中定义了如下依赖：

```
<dependency>
<groupId>org.nlpcn</groupId>
<artifactId>nlp-lang</artifactId>
<version>1.7</version>
<scope>compile</scope>
</dependency>
```



```

<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>1.7.21</version>
</dependency>

```

从上面我们不难看出 Ansj5.0.2 依赖 nlp-lang-1.7.jar 和 slf4j-api-1.7.21.jar 这两个 jar 包。因此你可以额外手动导入 nlp-lang-1.7.jar 和 slf4j-api-1.7.21.jar，为了方便用户使用 Ansj，作者提供了一个 all-in-one。其实作者就是把这些 jar 包的 class 文件打成了一个 jar 包，纯粹是为了照顾新手。Ansj 其他版本的 jar 包依赖同理。

如果项目中使用了 Maven，那么导入 Ansj 的 jar 包就容易多了，你只需要在 pom.xml 中添加如下配置即可：

```

<!--Ansj 的 maven 源，必须添加 -->
<repositories>
<repository>
<id>mvn-repo</id>
<url>http://maven.nlpcn.org/</url>
</repository>
</repositories>
<dependencies>
....
<dependency>
<groupId>org.ansj</groupId>
<artifactId>ansj_seg</artifactId>
<version>5.0.2</version>
</dependency>
....
</dependencies>

```

学习如何使用 Ansj 分词器最好的方法是从 Ansj 的 Github 上下载一份它的源码，然后导入到 IntelliJ-IDEA 或 Eclipse 中，在 org.ansj.test 代码测试包下，你会看到作者提供了大量测试用例，是大家学习 Ansj 的好资料，首先需要阅读 org.ansj.test.IndexAnalysisTest 类，其中演示了 Ansj 实现分词的两种用法：

```
System.out.println(IndexAnalysis.parse("北京地铁"));
```

或者直接新建一个 IndexAnalysis：

```
IndexAnalysis indexAnalysis = new IndexAnalysis(new InputStreamReader(new FileInputStream("D:/xxxxxx.txt")));
```

```

Term term = null;
while((term = indexAnalysis.next()) != null){
// 具体请参阅 Term 类的源码
    System.out.println "[" + term.getName() + "]/" + term.getNatureStr() + ":" +
        term.getOffe() + "->" + term.toValue());
}

```

如果你待分词的文本直接就是一个 String 类型字符串, 那么此时需要将 String 转换成 java.io.Reader 类型, 比如这样:

```
Reader reader = new BufferedReader(new StringReader(text));
```

IndexAnalysis 类的构造函数声明如下:

```

public IndexAnalysis(Forest... forests) {
    if (forests == null) {
        forests = new Forest[] { UserDefineLibrary.FOREST };
    }
    this.forest = forests;
}

public IndexAnalysis(Reader reader, Forest... forests) {
    this.forest = forests;
    super.resetContent(new AnsjReader(reader));
}

```

其中 forest 参数是可变参数, 它可以是零个或多个, 主要用于定义用户自定义字典文件, 即你可以通过编码的方式指定多个自定义字典文件的加载路径, 示例代码如下所示:

```

// key 即字典的唯一标识字符串, 后续可以通过 MyStaticValue.DIC.get(key)
// 返回一个 Forest 字典对象, dicPath 就是用户自定义扩展字典文件的绝对路径
MyStaticValue.initForest(String key, String dicPath)
或者
Forest forest = new Forest();
UserDefineLibrary.loadLibrary(forest, dicPath);
MyStaticValue.DIC.put(key, forest);
或者
Forest forest = Library.makeForest("C:/xxxxx/userLibrary.dic");
MyStaticValue.DIC.put(key, forest);

```

第一个参数 Reader 即需要分词的输入流。当然输入流 Reader 也可以不用在构造函数里指定, 但你没有在构造函数里指定输入流 Reader, 那么你需要在 IndexAnalysis 对象构建完毕后, 通过 this.resetContent (AnsjReader); 来设置输入流, 不过此时你需要传入一个 AnsjReader 对象, 然而 AnsjReader 就是 java.io.Reader 的一个子类, 构造 AnsjReader 时需要传入一个 java.io.Reader 对象, 这典型的装饰者模式啊。

```

public AnsjReader(Reader in) {
    this(in, defaultCharBufferSize);
}

public AnsjReader(Reader in, int sz) {

```

```

super(in);
if (sz <= 0)
    throw new IllegalArgumentException("Buffer size <= 0");
this.in = in;
cb = new char[sz];
}

```

如果你嫌太繁琐，那就直接 `IndexAnalysis analyzer = new IndexAnalysis();`

如果你想要在 Lucene 中使用 Ansj，那么你还需要访问 <http://maven.nlpcn.org/org/ansj/> 下载 `ansj_lucene5_plug-version.jar`。

下面是 AnsjAnalyzer 在 Lucene 5.x 中的使用示例代码：

```

Analyzer ca = new AnsjAnalyzer(TYPE.query);
String content = "这里是待分词的文本内容";
TokenStream ts = null;
try {
    ts = ca.tokenStream(content, new StringReader(content));
    OffsetAttribute offset = ts.addAttribute(OffsetAttribute.class);
    CharTermAttribute term = ts.addAttribute(CharTermAttribute.class);
    TypeAttribute type = ts.addAttribute(TypeAttribute.class);
    // 重置 TokenStream (重置 StringReader)
    ts.reset();
    // 迭代获取分词结果
    while (ts.incrementToken()) {
        System.out.println(offset.startOffset() + " - " + offset.endOffset() +
            " : " + term.toString() + " | " + type.type());
    }
    // 关闭 TokenStream
    ts.end();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    // 释放 TokenStream 的所有资源
    if (ts != null) {
        try {
            ts.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```

这样分词器是正常工作了，但是分词效果不太好，主要是因为我们还没有添加 Ansj 配置文件。下面来说说 Ansj 的配置文件，首先最重要的就是 `library.properties`，你可以从 Ansj 的源码包里获取到。通过该配置文件你可以配置用户自定义扩展字典文件的加载路径、歧义纠正字典文件的加载路径等。Ansj 分词器的配置参数可以定义在 `properties` 属性文件中，当 `MyStaticValue` 类被加载时，会尝试加载 Ansj 分词器 `properties` 属性文件。该属性文件加载

逻辑如下:

首先尝试在当前 Classpath 下加载 `ansj_library_语言代码_国家代码.properties` 文件, 如果当前处于中文操作系统, 那么就是加载 `ansj_library_zh_CN.properties`。

如果没有找到该文件, 会继续依次尝试在当前项目根目录下和 `java.library.path` `java.class.path` 以及三个路径下加载 `ansj_library.properties` 属性文件。

如果没有找到 `ansj_library.properties` 属性文件, 会继续在当前 Classpath 下加载 `library_语言代码_国家代码.properties` 文件, 如果当前处于中文操作系统, 那么就是加载 `library_zh_CN.properties`。

如果还是没有找到 `library_zh_CN.properties` 属性文件, 会继续依次尝试在当前项目根目录下和 `java.class.path` 以及 `java.library.path` 三个路径下加载 `library.properties` 文件。

最后还是没有找到的话, 那么它就放弃了, 并打印 Log 警告信息, 内容如下:

```
not find library.properties in classpath use it by default !
```

表示 `library.properties` 属性文件没有找到, 会使用默认值。

如果最后加载到了 `library.properties` 或 `ansj_library.properties` 文件, 那么会解析 `properties` 文件并获取该文件中 `key=dic` 的配置项的值即用户配置的自定义字典文件加载路径, 然后将该路径以 “dic” 为 key 值放入一个名称为 DIC 的 map 集合中。如果用户在 `properties` 文件中没有配置 `dic=/xxxxxx/xxxxxx.dic`, 那么会使用默认值 `dic=library/default.dic`。

当 `UserDefineLibrary` 类被加载时, 会在其自身的静态代码块中执行 `initUserLibrary` 方法, 该方法会从 `MyStaticValue` 的 DIC map 中获取 `key=dic` 的用户自定义字典文件加载路径, 然后将该路径值传入 `UserDefineLibrary.loadLibrary` 方法中执行用户自定义字典文件的加载, 加载过程大致如下:

- 1) 如果用户配置的是字典文件的绝对路径, 那么会直接通过 `new File (path)` 这种方式加载;
- 2) 如果用户配置的是字典文件存放目录的绝对路径, 那么会尝试加载该目录下的所有文件, 并找到所有以 `.dic` 为后缀名的文件, 作为用户的自定义字典文件;
- 3) 如果用户配置的是字典文件的相对路径, 那么会尝试使用 `new File (path)` 方式进行加载。如果通过 `new File (path)` 方式还是找不到, 那么再尝试从 Classpath 路径下通过当前类加载器去加载, 实现代码为:

```
URL url = UserDefineLibrary.class.getClassLoader().getResource(path);
```

相对路径的方式会随着你代码的运行环境不同而改变, 所以, 我建议自定义扩展字典文件的加载路径还是配置成绝对路径吧, 这样省心多了。

- 4) 通过 `url.getPath()` 可以获取到绝对路径, 然后重复前面 1)、2) 步骤。如果用户配置的是一个无效的路径, 那么最终会导致用户自定义字典文件加载失败, 并打印如下警告信息:

```
dic dic not found in config
```

同理，UserDefineLibrary 类在加载时，也会尝试去加载 ambiguityLibrary 歧义词纠正字典文件，加载逻辑大体上与用户自定义字典文件一致。

通过上面我们得知，关于 Ansj 需要的 properties 属性文件放置路径问题，你可以直接在项目根目录下放置一个 ansj\_library.properties 或 library.properties 文件，你也可以直接下项目的 src（对于 Maven 项目，那就是 src/main/java）下放置一个 ansj\_library\_zh\_CN.properties 或 library\_zh\_CN.properties 文件（对于英文操作系统环境下，zh\_CN 就要替换成 en\_US 啦）。关于用户自定义扩展字典文件和自定义歧义词纠正字典文件放置路径问题，你可以直接配置成字典文件的硬盘绝对路径或者字典文件存放目录的硬盘绝对路径，如果配置成目录的硬盘绝对路径，那么该目录下所有文件名称以 .dic 结尾的文件都会被当作字典文件。你也可以配置成字典文件的相对路径或者字典文件存放目录的相对路径，这里的相对路径对于 Solr 来讲首先会被理解为相对于项目的根目录，如果没找到再理解为相对于当前 Classpath。



**注意** 这里都是以 Ansj-5.0.3 为例进行说明，其他版本中配置文件的加载逻辑可能会有所出入，具体请以各版本源码为准。

library.properties 提供了如下几个配置项：

```
# 用户自定义扩展字典文件的加载路径
dic=library/default.dic
# 自然语言模型文件的加载路径
#crf_dic1=library/crf.model
# 用户自定义歧义词纠正字典文件的加载路径
ambiguityLibrary=library/ambiguity.dic
# 是否启用 realName，realName 就是是否根据 token 的偏移量来纠正 token 的正确文本值
# 保持默认即可
isRealName=true
# 是否开启人名自动识别功能，默认值为 true 即开启，保持默认即可
isNameRecognition=true
# 是否开启数字自动识别功能，默认值为 true 即开启，保持默认即可
isNumRecognition=true
# 是否开启阿拉伯数字与单位自动合并功能，默认值为 true 即开启，保持默认即可
isQuantifierRecognition=true
```

其实 library.properties 属性配置文件并不是必须的，你完全可以通过编程的方式进行配置，比如配置用户自定义扩展字典文件的加载路径：

```
MyStaticValue.DIC.put(MyStaticValue.DIC_DEFAULT," D:/xxx/userdefine.dic");
```

比如配置用户自定义扩展字典优先：

```
MyStaticValue.isSkipUserDefine = false;
```

比如配置用户自定义歧义词纠正字典文件的加载路径：

MyStaticValue. ambiguityLibrary = "library/ambiguity.dic";

更详细的配置项可以通过查阅 MyStaticValue 类源码里的静态变量来了解，如下所示：

```
// 是否开启人名识别
public static Boolean isNameRecognition = true;
// 是否开启数字识别
public static Boolean isNumRecognition = true;
// 是否数字和量词合并
public static Boolean isQuantifierRecognition = true;
// 是否显示真实词语
public static Boolean isRealName = false;
/**
 * 当用户自定义扩展字典与默认字典拥有相同的词时，
 * 是否优先以用户自定义扩展字典里的词为准
 * 配置为 true 即表示用户自定义扩展字典里的词优先
 */
public static boolean isUserDefinePrefer = true;
/**
 * 用户自定义扩展词典的加载，如果是路径就扫描路径下的所有 .dic 文件
 */
public static String userDefineLibrary = "library/default.dic";
/**
 * 用户自定义歧义纠正词典的加载，如果是路径就扫描路径下的所有 .dic 文件
 */
public static String ambiguityLibrary = "library/ambiguity.dic";
// 用户自定义词典
public static final Map<String, Object> DIC = new HashMap<String, Object>();
```

将源码中原来的 isSkipUserDefine 变量名称修改为 isUserDefinePrefer，我觉得这样更容易让人理解，isSkipUserDefine 原意是当用户自定义字典里的词与核心字典里的词重复了，是否以核心字典为准，我将含义颠倒过来了，将其命名为 isUserDefinePrefer，即当两者的词重复时，优先以用户自定义字典里的词为准。userDefineLibrary 变量也是我自己新增的，主要是由于默认添加用户自定义字典文件只能通过 MyStaticValue.DIC.put(key, "xxxx/xxxxxx.dic"); 这种方式来设置，我嫌太繁琐，所以定义了一个 userDefineLibrary 静态变量，这样用户只需要 MyStaticValue.userDefineLibrary 这种方式设置自定义字典文件的加载路径即可。当然，光添加一个 userDefineLibrary 静态变量还不够，你还需要修改 MyStaticValue 的 getDicForest 方法，修改如下：

```
/**
 * 根据模型名称获取 crf 模型
 * @param key
 * @return
 */
public static Forest getDicForest(String key) {
    Object temp = DIC.get(key);
    if (temp == null) {
```

```

// 如果是用户自定义扩展字典的 key, 并且用户设置了 userDefineLibrary
// 那么根据用户设置的 userDefineLibrary 路径去加载用户自定义扩展字典文件
if(key.equals(USER_DEFINE_DIC_KEY)
&& !StringUtil.isBlank(userDefineLibrary)) {
    DIC.put(key,userDefineLibrary);
    return initForest(USER_DEFINE_DIC_KEY, userDefineLibrary);
}
LIBRARYLOG.warn("dic {} not found in config ",key);
return null;
} else if (temp instanceof String) {
    return initForest(key, (String) temp);
} else {
    return (Forest) temp;
}
}
}

```

另外 MyStaticValue 里的 isSkipUserDefine 并没有提供在 library.properties 中进行配置, 你只能通过 MyStaticValue 类来设置, 因此你可以对 MyStaticValue 类的 static 静态代码块源码做如下修改:

```

else if (key.startsWith("crf_")) {
    // 省略.....
} else if (key.equals(USER_DEFINE_PREFER_KEY)) {
    // 获取 properties 配置文件里的 user_define_prefer
    isUserDefinePrefer = !(Boolean.valueOf(rb.getString(key)));
}
}

```

这样你就可以在 library.properties 属性文件中添加 user\_define\_prefer=true 配置项进行参数设置。当然上述这些源码级的修改并不是必须的, 请读者自行选择, 这里仅仅是为了方便大家更好的使用 Ansj 分词器提出了一点小看法。毕竟修改源码还得重新编译打 jar 包, 比较繁琐。

了解了 library.properties 文件应该如何放置以及如何配置, 那么接下来就是按照配置内容将相应的字典文件放置到配置好的目录下。字典文件可以从源码包里获取到, 这里我建议直接复制源码里提供的字典文件作为模板, 然后进行修改, 不太建议自己新建 dic 文件, 因为 dic 字典文件要求是 UTF-8 编码且无 BOM 格式, 自己新建可能会出现字典文件编码问题。一般我们需要配置两个字典文件: 用户自定义扩展字典和用户自定义歧义纠正字典。

用户自定义扩展字典主要用于定义一些 Ansj 分词器不认识的词语, 比如网络流行用语: 么么哒、不明觉厉、城会玩、颜值、狗带等。定义格式如下所示:

```

么么哒 a 1000
城会玩 a 1001

```

字典的每一行按 tab 键分 3 列, 3 列依次表示词语、词性、词频。关于词性表示方法, 请参阅“词性对照说明中科院版本”, 这个通过网络能搜索到, 由于内容 100 多行这里不便一一列举。



用户自定义歧义纠正字典用于解决一些具有分词歧义的词语分词问题，用户可以通过此字典文件告诉 Ansj 分词器该如何切分它，举个例子，比如“动漫游戏”，这里“漫游”也是一个词语，但在这里显然应该分成“动漫”和“游戏”，因为毕竟分词器不比人脑，人脑具有根据语境进行语义分析，而分词器还没那么智能，所以这种情况下，你需要在歧义纠正字典里进行这些特殊情况下的分词切分定义，定义格式如下所示：

```
动漫 n 游戏 n
邓颖超 nr 生前 t
```

字典的每一行依次表示词语 \t 词性 \t 词语 \t 词性词语 \t 词性 \t……通过词性和 \t 符号将词语分割开，这样就告诉了 Ansj 分词器在这种具有歧义的情况下该如何按照你定义的规则去分词。

默认 Ansj 分词器是不带停用词功能的，如果需要添加停用词字典，你需要导入 Ansj 与 Lucene 兼容的插件 jar 包即 ansj-lucene5-plugin-version.jar。你可以在创建 AnsjAnalyzer 分词对象时，通过它的构造函数传入一个停用词字典文件路径或者一个包含多个停用词的 Set 集合。示例代码如下：

```
Analyzer ca = new AnsjAnalyzer(Type.query, "xxxx/stopwords.dic");
或者
Set<String> stopwords = new HashSet<String>();
stopwords.add("the");
stopwords.add("is");
……// 省略
Analyzer ca = new AnsjAnalyzer(Type.query, stopwords);
```

那么这个停用词字典文件的加载路径应该如何指定呢？通过阅读源码发现，停用词字典文件是在 AnsjAnalyzer 类的 filter 方法里完成加载并读取转换成 Set<String>。内部本质是通过 new File(path) 这种方式来加载停用词字典文件的，此时假如用户传入的 path 值是一个硬盘绝对路径，那大家都懂的。但是假如用户传入的 path 值是一个相对路径，比如 path=library/stopwords.dic，那么此时就是相对于当前项目的根目录而言，假设你当前项目所在目录为 E:/javaWork/ansj-demo，那么实际完整加载路径就是 E:/javaWork/ansj-demo/library/stopwords.dic，也就是说此时你必须将停用词字典文件放置在项目 ansj-demo 的根目录下才能成功被加载，并且此时 path 值的开头不能带有斜杠 /。总而言之，停用词字典文件的加载路径你设置为绝对路径就万事大吉了，如果设置为相对路径，而你又不了解内部这些细节，那么就是自己往坑里跳。Ansj 分词器在字典文件加载路径处理方面没有统一，如果不阅读源码去了解，容易让使用 Ansj 分词器的新手们感觉发蒙啊！在这点上，Ansj 确实不太讲究。我觉得还不如直接统一从 ClassPath 下加载，比如这样实现：

```
InputStream is =
AnsjAnalyzerTest.class.getClassLoader().getResourceAsStream(path);
BufferedReader reader = new BufferedReader(new InputStreamReader(is, "UTF-8"));
```

此时假如 `path=library/stopwords.dic`, 那么就会从当前项目的 classpath (即 `src` 或 `src/main/java` 下) 下载字典文件, 假如 `path=/library/stopwords.dic`, 此时前面的斜杠 / 就代表项目根目录。但这种方式也有弊端: 只能使用相对路径, 无法使用绝对路径。

默认 `ansj-lucene5-plugin-version.jar` 里并没有包含了 `AnsjTokenizerFactory` 的实现, 如果你需要在 Solr 中使用 Ansj 分词, 你除了需要导入 `ansj-lucene5-plugin-version.jar`、`ansj_seg-5.0.2.jar`、`nlp-lang-1.7.jar`, 然后还需要自定义一个 `AnsjTokenizerFactory`, 最后打成 jar 包放到 `core/lib` 目录下, 这里我提供一个我自定义的 `AnsjTokenizerFactory` 实现, 仅供参考, 代码如下所示:

```
import org.apache.lucene.analysis.Tokenizer;
import org.apache.lucene.analysis.util.TokenizerFactory;
import org.apache.lucene.util.AttributeFactory;
import java.util.Map;

public class AnsjTokenizerFactory extends TokenizerFactory {
    /** 分词器类型, 可选值有: base, index, query, to, user, search */
    private String analyzerType;
    /** 自定义停用词词典文件加载路径 */
    private String stopwordsDicPath;

    public AnsjTokenizerFactory(Map<String, String> args) {
        super(args);
        analyzerType = get(args, "analyzerType", "query");
        stopwordsDicPath = get(args, "stopwords");
    }

    @Override
    public Tokenizer create(AttributeFactory factory) {
        return new AnsjTokenizer(factory,
            AnalyzerType.TO.convertToAnalyzerType(this.analyzerType),
            stopwordsDicPath);
    }
}
```

然后就是将 `library.properties` 或 `ansj_library.properties` 属性文件复制到 ClassPath 下即 `E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes` 目录下 (`classes` 目录若不存在请手动创建), 这里的 `E:\apache-tomcat-7.0.55` 即你的 Tomcat 安装根目录。然后 `ansj_properites` 属性文件配置示例如下:

```
# 用户自定义扩展字典文件加载路径
dic=library/ext.dic
# 用户自定义歧义纠正字典文件加载路径
ambiguityLibrary=library/ambiguity.dic
```

接下来你需要在 `E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes` 目录下创建一个 `library` 目录, 用于存放 `ansj_properites` 属性文件中配置的两个字典文件。然后将两个字典文件复制到刚刚新建的 `library` 目录下。然后在 `schema.xml` 里定义 `fieldType`, 示例如下所示:

```

<field name="content" type="text_anjs" indexed="true" stored="true" omitNorms=
"true" multiValued="false"/>
<fieldType name="text_anjs" class="solr.TextField">
<analyzer>
<tokenizer class="org.apache.lucene.util.AnsjTokenizerFactory"
analyzerType="query" stopwords="D:/dic/stopwords.dic"/>
</analyzer>
</fieldType>

```

然后你需要将 stopwords 参数配置的停用词字典文件 stopwords.dic 放置到 D:/dic 目录下，最后重新启动 Tomcat，这样 Ansj 分词器就配置好了。套路就那么几个，重点是要了解各个配置文件及字典文件的放置路径，否则配置文件会加载不到，这样配置就无法生效了。

### 4.3.3 MMSeg4J 分词器

MMSeg4J 是基于 Chih-Hao Tsai 的 MMSeg 算法 (<http://technology.chtsai.org/mmseg/>) 实现的一款 Java 中文分词器，并实现了 Lucene 的 Analyzer 和 Solr 的 TokenizerFactory 以方便兼容 Lucene 和 Solr。MMSeg 算法有两种分词方法：Simple 和 Complex，都是基于正向最大匹配。Complex 加了四个规则过滤。官方称：词语的正确识别率达到了 98.41%。MMSeg4J 实现了这两种分词算法。

MMSeg4J 的源代码是托管在 google code 上的，访问地址是 <http://code.google.com/p/mmseg4j/>。我已经将源代码上传到 Github 上，你可以访问如下地址自由下载：

<https://github.com/yida-lxw/mmseg4j-1.9.2>

执行 `git clone https://github.com/yida-lxw/mmseg4j-1.9.2.git` 命令就能克隆到你本地啦。源码是基于 Maven 构建，所以如果你想要导入源码到 IDEA 或 Eclipse 中，你必须先本地安装好 Maven 环境。MMSeg4J 源码包含了 3 个子模块：mmseg4j-core、mmseg4j-analysis、mmseg4j-solr。其中 mmseg4j-core 是实现分词的核心模块，mmseg4j-analysis 是为了兼容 Lucene 提供 Analyzer 实现类的模块，同理，mmseg4j-solr 是为了兼容 Solr 提供 TokenizerFactory 实现类的模块。如果你对 MMSeg4J 源代码感兴趣，你可以将其导入 IDEA 或 Eclipse 中，如果你想要在 Lucene 或 Solr 中使用它，你需要将各个模块打包成 jar 包。

Maven 打包命令使用：`package -Dmaven.test.skip=true`

打包后的 jar 包放置在各个模块的 target 目录下。如果你想要学习如何使用 MMSeg4J 分词器，那么可以查看 mmseg4j-analysis 模块的测试包 src/test/java 下的 MMSegAnalyzerTest 类，该类详细介绍了 SimpleAnalyzer、MaxWordAnalyzer 和 ComplexAnalyzer 三个分词器的使用。三者分别对应了 3 种分词模式：simple、complex、maxword。3 个分词器均继承自 MMSegAnalyzer 基类。创建 MMSeg4J 的分词器最简单的方式就是直接 `new XXXAnalyzer()` 即可。但分词器默认还提供了另一个构造函数 `public MMSegAnalyzer(String dicPath,String`

mode,String stopwordsPath), 其中 mode 参数就是表示 3 种分词模式, 此外用户可以传入两个字典的加载路径, 第一个 dicPath 即 MMSEg4J 分词器内部需要的 3 个字典文件的存放目录路径: chars.dic、words.dic、units.dic, 依次表示单字词库、汉语词语词库和单位词库(单位都是单个字)。第 3 个参数 stopwordsPath 即停用词字典文件的加载路径。想要明白如何指定这些字典文件的加载路径, 首先有必要熟悉它内部的字典文件加载逻辑。字典文件的加载是由 Dictionary 类负责, Dictionary 类是单实例模式设计, 用户可以通过 getInstance(path) 方法来构建 Dictionary 字典对象。这里分两种情况: 用户传入和未传入自定义扩展字典文件加载路径。若用户未传入, 那么会调用 getDefaultPath() 方法获取字典文件的默认加载路径, 具体获取逻辑如下:

- ❑ 首先通过 System.getProperty("mmseg.dic.path") 获取系统属性 mmseg.dic.path 值, 用户可以通过 System.setProperty("mmseg.dic.path","xxxxx") 或者 -D mmseg.dic.path=xxxx 方式设置;

- ❑ 如果用户没有设置, 那么会通过当前类加载器从 CalssPath 下查找 data 目录, 具体代码如下:

```
Dictionary.class.getClassLoader().getResource("data");
```

- ❑ 如果在 ClassPath 下没找到该 data 目录, 那么会继续在当前工作目录(即当前项目的根目录)下, 继续查找 data 目录, 具体代码如下:

```
defPath = System.getProperty("user.dir") + "/data";
```

- ❑ 如果还是没找到, 那么就放弃了, 随即抛出警告信息, 提示默认字典文件不存在, 否则就返回默认字典文件的加载路径。

如果用户传入了自定义扩展字典文件的加载路径, 那么就以用户提供的为准, 否则以 getDefaultPath() 方法返回的默认加载路径为准。获取到字典的加载路径后, 通过 normalizeFile() 方法对路径进行规范化, 比如相对路径转换成绝对路径。首先通过 dics, get() 从缓存中获取字典对象, 如果找到了则直接返回, 否则根据字典加载路径通过 new Dictionary 方式从硬盘上加载字典。Dictionary 类的构造函数里通过自身的 init(path) 方法完成字典文件的加载, 而 init 方法内部又是调用 reload() 方法实现字典文件加载。依赖的 3 个字典文件通过 loadDic() 和 loadUnit() 调用完成加载, loadDic 完成对 chars.dic 和 words/dic 的加载, loadUnit 完成对 units.dic 的加载。loadDic 的字典加载逻辑大致如下:

假如用户传入的是字典文件存放目录的硬盘绝对路径, 那么就在该目录下查找 data 目录, 如果 data 目录存在, 那么继续在 data 目录下查找 chars.dic 字典文件, 如果 data 目录不存在, 那么就会继续在 ClassPath 下查找 data/chars.dic, 具体执行代码为:

```
// 适合于文件在 jar 包内部的情况
this.getClass().getResourceAsStream("/data/chars.dic");
```

加载 ClassPath 下的资源文件的另一种方式如下:

```
// 适合于文件在 jar 包外部的 classpath 下
this.getClass().getClassLoader().getResourceAsStream("data/chars.dic");
```

假如用户传入的是字典文件存放目录的相对路径，此时是采用 `new File(path)` 方式加载的，然后通过 `File` 类的 `getCanonicalFile()` 方法将用户传入的相对路径转换成绝对路径，之后再在该路径下查找 `data/chars.dic`。如果仍然找不到，那么会继续在 `ClassPath` 下查找 `data/chars.dic`。

加载 `units.dic` 单位字典文件的逻辑与之类似，加载用户 `words.dic` 稍微有点不同，除了会加载默认的 `words.dic` 字典以外，还会加载 `dicPath` 目录下用户自定义的扩展字典，用于自定义一些分词器不认识的新词，比如网络流行词。不过，`MMSeg4J` 对用户自定义的 `words.dic` 的文件名称有严格约束，即必须是以 `words` 开头和以 `.dic` 为后缀名的文件，具体实现代码如下：

```
protected File[] listWordsFiles() {
    return dicPath.listFiles(new FilenameFilter() {
        public boolean accept(File dir, String name) {
            return name.startsWith("words") && name.endsWith(".dic");
        }
    });
}
```

如果你想要覆盖 `MMSeg4J` 默认的 3 个字典文件，那么你可以指定 `dicPath` 参数，然后将你自定义的字典文件放置在 `dicPath` 参数指定的目录下即可，这里建议你将在 `dicPath` 参数指定为绝对路径比如 `E:/dic/data/`，因为相对路径容易让人发蒙。

至于 `stopwordsPath` 参数，表示停用词字典文件的加载路径，是我额外扩展的，`MMSeg4J` 自身是不支持停用词过滤功能的。停用词字典文件的加载实现方式我是这样实现的：

```
this.getClass().getClassLoader().getResourceAsStream(dir);
```

即从 `ClassPath` 下加载用户指定的字典文件，此时 `stopwordsPath` 参数你必须指定为相对路径，比如 `stopwordsPath=stopwords.dic`，那么此时 `ClassPath` 即相当于 `solr` 项目的 `WEB-INF/classes` 目录，假定你的 `Tomcat` 安装根目录是 `E:\apache-tomcat-7.0.55`，那么此时 `ClassPath` 的完整绝对路径就是 `E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes`，你只需要将 `stopwords.dic` 停用词字典文件放置在 `ClassPath` 下即可。

如果你想要在 `Lucene` 中使用 `MMSeg4J`，那么首先需要导入 `mmseg4j-core-1.9.2.jar` 和 `mmseg4j-analysis-1.9.2.jar` 两个 `jar` 包。然后创建 `MMSegAnalyzer` 的任意一个子类对象。如果你需要使用停用词功能，那么还需要在构造函数里传入停用词字典文件的加载路径。如果你需要扩展默认的 3 个字典文件，那么则需要传入一个 `dicPath` 参数，用于指定这 3 个字典文件的存放路径，使用示例代码如下所示：

```
Analyzer analyzer = new SimpleAnalyzer("data", "mmeseg4j/stopwords.dic");
```

由于 MMSeg4J 默认不支持停用词功能, 所以我对源码进行了部分修改, 这里贴出部分关键性代码:

```
public class MMSegTokenizer extends Tokenizer {
    private MMSeg mmSeg;
    private CharTermAttribute termAtt;
    private OffsetAttribute offsetAtt;
    private PositionIncrementAttribute positionAtt;
    private TypeAttribute typeAtt;
    /** 自定义停用词 */
    private Set<String> filter;
    public MMSegTokenizer(AttributeFactory factory, Seg seg, String stopwordsDir)
    {
        super(factory);
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
        addStopwords(stopwordsDir);
    }

    public MMSegTokenizer(AttributeFactory factory, Seg seg, Set<String> filter)
    {
        super(factory);
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
        this.filter = filter;
    }

    public MMSegTokenizer(AttributeFactory factory, Seg seg) {
        super(factory);
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
    }

    /**
     * 构造函数
     * @param seg
     * @param filter 自定义停用词 Set
     */
    public MMSegTokenizer(Seg seg, Set<String> filter) {
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
    }
}
```



```

        this.filter = filter;
    }
    /**
     * 构造函数
     * @param seg
     * @param stopwordsDir 自定义停用词词典文件加载路径
     */
    public MMSegTokenizer(Seg seg, String stopwordsDir) {
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
        addStopwords(stopwordsDir);
    }
    public MMSegTokenizer(Seg seg) {
        mmSeg = new MMSeg(input, seg);
        termAtt = addAttribute(CharTermAttribute.class);
        offsetAtt = addAttribute(OffsetAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        positionAtt = addAttribute(PositionIncrementAttribute.class);
    }
    public void reset() throws IOException {
        super.reset();
        mmSeg.reset(input);
    }
    @Override
    public final boolean incrementToken() throws IOException {
        clearAttributes();
        Word word = null;
        String wordStr = null;
        boolean flag = true;
        int position = 0;
        do {
            word = mmSeg.next();
            if (null == word) {
                break;
            }
            wordStr = word.getString();
            if (filter != null && filter.contains(wordStr)) {
                continue;
            } else {
                position++;
                flag = false;
            }
        } while (flag);
        if (null != wordStr) {
            positionAtt.setPositionIncrement(position);
            termAtt.copyBuffer(word.getSen(), word.getWordOffset(), word.getLength());
            offsetAtt.setOffset(word.getStartOffset(), word.getEndOffset());
            typeAtt.setType(word.getType());

```



```

        return true;
    }
    end();
    return false;
}

/**
 * 添加停用词
 * @param dir
 */
private void addStopwords(String dir) {
    if (dir == null || "".equals(dir)) {
        return;
    }
    this.filter = new HashSet<String>();
    InputStreamReader reader;
    try {
        InputStream is = this.getClass().getClassLoader().getResourceAsStream(dir);
        reader = new InputStreamReader(is, "UTF-8");
        BufferedReader br = new BufferedReader(reader);
        String word = br.readLine();
        while (word != null) {
            this.filter.add(word);
            word = br.readLine();
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException("No custom stopwords file found");
    } catch (IOException e) {
        throw new RuntimeException("Custom stopwords file io exception");
    }
}
}

```

如果想要在 Solr 中使用 MMSeg4J，那么你还需要额外添加 mmseg4j-solr-1.9.2.jar。然后将停用词字典文件放置在 classpath 下。由于默认 MMSeg4J 不支持停用词，所以我对 MMSegTokenizerFactory 类的源码也进行了改造，具体代码如下所示：

```

public class MMSegTokenizerFactory extends TokenizerFactory implements Resource-
LoaderAware {
    static final Logger log = Logger.getLogger(MMSegTokenizerFactory.class.getName());
    private Dictionary dic = null;
    /** 自定义停用词加载路径 */
    protected String stopwordsPath;

    public MMSegTokenizerFactory(Map<String, String> args) {
        super(args);
        this.stopwordsPath = get(args, "stopwordsPath");
    }

    private Seg newSeg(Map<String, String> args) {

```

```

    Seg seg = null;
    String mode = args.get("mode");
    if("simple".equalsIgnoreCase(mode)) {
        log.info("use simple mode");
        seg = new SimpleSeg(dic);
    } else if("complex".equalsIgnoreCase(mode)) {
        log.info("use complex mode");
        seg = new ComplexSeg(dic);
    } else if("maxword".equalsIgnoreCase(mode)) {
        log.info("use max-word mode");
        seg = new MaxWordSeg(dic);
    } else {
        log.info("use max-word mode as default.");
        seg = new MaxWordSeg(dic);
    }
    return seg;
}

@Override
public void inform(ResourceLoader loader) {
    String dicPath = getOriginalArgs().get("dicPath");
    dic = Utils.getDict(dicPath, loader);
    log.info("loading dictionary ... in path [" + dic.getDicPath().toURI() + "]");
}

@Override
public Tokenizer create(AttributeFactory factory) {
    if(null == stopwordsPath || "".equals(stopwordsPath)) {
        return new MMSegTokenizer(factory, newSeg(getOriginalArgs()));
    }
    return new MMSegTokenizer(factory, newSeg(getOriginalArgs()), stopwordsPath);
}
}

```

改造后的代码我已经上传到 Github 仓库，更详细代码请访问我的 Github 下载获取：

<https://github.com/yida-lxw>。

然后你需要在 schema.xml 中配置 fieldType，配置示例如下所示：

```

<fieldType name="text_mmseg4j" class="solr.TextField">
<analyzer>
<tokenizer class="com.chenlb.mmseg4j.solr.MMSegTokenizerFactory"
    dicPath="D:/data" mode="complex" stopwordsPath="stopword.dic"/>
</analyzer>
</fieldType>

```

此外 MMSeg4J 默认不能切分数字与英文连接在一起的字符串，比如 160cm、80KG 等，默认会被当作整体切分成一个 Token，如果想要切分成 160 和 cm 两个 Token，你可以再配置一个 MMSeg4J 提供的 CutLetterDigitFilterFactory，它支持对数字与英文字母混合文本切分，配置示例如下所示：

```

<fieldType name="text_mmseg4j" class="solr.TextField">
<analyzer>

```

```
<tokenizer class="com.chenlb.mmseg4j.solr.MMSegTokenizerFactory"
          dicPath="D:/data" mode="complex" stopwordsPath="stopword.dic"/>
<filter class="com.chenlb.mmseg4j.solr.CutLetterDigitFilterFactory"/>
</analyzer>
</fieldType>
```

### 4.3.4 Paoding 分词器

庖丁 (Paoding) 中文分词器是使用 Java 开发的, 可结合到 Lucene 应用中, 为互联网、企业内部网使用的中文搜索引擎分词组件。Paoding 填补了国内中文分词方面开源组件的空白, 致力于此并希冀成为互联网网站首选的中文分词开源组件。Paoding 中文分词追求分词的高效率和用户良好体验。它具有如下特性:

- Paoding's Knives (庖丁解牛) 中文分词器具有高扩展性: 能够非常方便的扩充字典, 也可以非常方便的添加停用词;
- 引入隐喻, 采用完全的面向对象设计, 构思先进;
- 拥有极高的分词效率和极高效率的字典查找算法, 尽量避免无谓试探查找;
- 算法简练 - 简单易理解的算法, 但效率却是非常高效的;
- 自定义扩展字典文件可以无限扩展, 且分类细致, 便于字典管理维护;
- 能够对未知的词汇进行合理解析。

Paoding 分词器项目源码起初是托管在 Google Code 上, 下载地址如下:

<http://code.google.com/p/paoding/>

由于 Google 在中国早就被和谐, 故已有国人将其源代码托管到 OSChina Git 上, 下载地址为: <http://git.oschina.net/zhzhenqin/paoding-analysis>

不过, 最后一次更新时间已经是 3 年前了, 还是基于 Lucene 4.x 构建的, 为此, 我修改了源码使其升级支持 Lucene&Solr 5.x。至于如何修改的, 只字片语难以言表, 有兴趣的童鞋请直接访问我的 Github 下载我修改升级后的源代码, 然后自行研究。我已经将代码上传到我的 Github 仓库, 下载地址为: <https://github.com/yida-lxw/paoding-analysis>。

获取到 Paoding 分词器的源码后, 你可以将其导入 IDEA 或 Eclipse 中, 然后对其进行编译打包, 最终你将在项目的 target 目录下得到一个 paoding-analysis.jar。如果你仅仅只是在 Lucene 中使用 Paoding 分词器, 那么你还不需要将打包后的 jar 包里的 solr 包删除, 因为 solr 包下的 PaodingTokenizerFactory 类是使用 Solr 时才需要的类, 如果不删除它, 你还需要额外导入 solr-core-version.jar。具体操作如图 4-5 所示。

当然比较优雅的方式是打包成两个 jar 包, solr 包

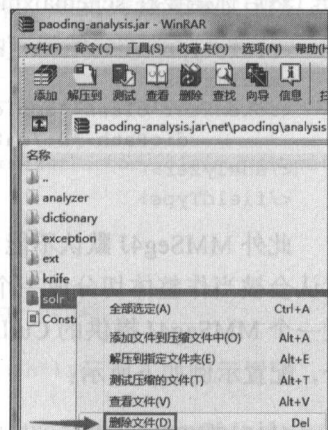


图 4-5 非 Solr 环境下删除 solr 包

下的 `PaodingTokenizerFactory` 类应该单独打成一个 jar 包，但由于仅仅只是一个类，我就偷个懒直接放在了一个项目里。你可以自己再创建一个 Java Project，然后导入 `solr-core-version.jar`，然后将 `solr` 包复制过去，这样以后单独打包供 Solr 使用，这个留给你们自己去完成。

在 Lucene 中使用 Paoding 分词器，首先你需要导入 `paoding-analysis.jar`，然后你需要配置 Paoding 的主配置文件 `paoding-analysis.properties`。`paoding-analysis.properties` 属性文件并不是必需提供的，因为它们已经包含在 `paoding-analysis.jar` 内部，但是通常建议还是再复制一份放到项目的 ClassPath 下，对默认 jar 包内部的 `properties` 配置进行覆盖，这样也便于后续对配置进行修改。主配置文件 `paoding-analysis.properties` 主要用于导入其他 `properties` 文件，这样便于分多个 `properties` 文件进行配置管理。其中 `paoding-analyzer.properties` 文件用于配置分词器模式以及 Paoding 字典文件编译器。`paoding-dic-home.properties` 用于配置 Paoding 分词器的自定义扩展字典文件的主目录即 `dic-home`，配置示例如下所示：

```
# 如果配置成 "system-env" 或 "this", 那么表示 dic-home 应该从系统环境变量
# 里去获取, 即此时你需要像配置 JAVA_HOME 一样, 配置一个 PAODING_DIC_HOME
#paoding.dic.home.config-first=system-env
#dic.home 表示庖丁解牛分词器的用户自定义扩展字典文件的主目录
# 如果配置成 classpath:dic, 则表示在 classpath 路径下查找 dic 目录,
# 当然你也可以配置成硬盘绝对路径
paoding.dic.home=dic
# 用于配置 dic-home 的绝对路径, 如果配置了此配置项, 那么上述 2 个配置项直接无视,
# 以 paoding.dic.home.absolute.path 配置为准
#paoding.dic.home.absolute.path=D:/dic/
# 用户自定义扩展字典文件更新自动检测间隔周期, 单位: 秒
# 即每间隔多长时间检查下 dic-home 目录下的字典文件更新了没有,
# 如果更新了, 则自动重新编译字典文件使其生效。
#paoding.dic.detector.interval=60
```

获取词典安装主目录配置（即 `dic.home`）的具体逻辑如下所示：

- 如果用户配置了 `paoding.dic.home.absolute.path`，则将其作为词典的安装主目录，如果没有配置，则继续看 `PAODING_DIC_HOME` 环境变量；
- 如果用户配置了 `PAODING_DIC_HOME` 环境变量，则将其作为词典的安装主目录，否则使用属性文件的 `paoding.dic.home` 配置；
- 但是如果属性文件中强制配置 `paoding.dic.home.config-first=this`，则优先考虑属性文件的 `paoding.dic.home` 配置；
- 此时只有当属性文件没有配置 `paoding.dic.home` 时才会采用环境变量的配置；
- 如果环境变量和属性文件都没有配置词典安装主目录，则尝试在当前目录和类路径 ClassPath 下查找是否存在 `dic` 目录，如果存在，则采纳它为 `paoding.dic.home`；如果尝试后均失败，则抛出 `PaodingAnalysisException` 异常。

字典文件的主目录配置好之后，你需要将自定义的扩展字典文件复制到该目录下，这

些字典文件你可以从 Paoding 的源码包里的 dic 目录下获取到。Paoding 分词器默认会自动加载并编译字典文件主目录下的所有 dic 字典文件，如果想要继续添加扩展新的字典文件，那么你需要如下几个步骤：

1) 首先在 dic.home 目录下新建一个 dic 字典文件，注意文件编码必须为 UTF-8 且无 BOM 格式。

2) 使用文本编辑软件（请不要使用系统自带的记事本进行编辑）打开并编辑，添加自定义词语，每个词语独占一行。

3) 添加或修改字典文件后必须删除 dic.home 目录下的 .compiled 文件夹。

如果你想要排除 dic.home 目录下部分字典文件不加载，那么你可以在 dic.home 目录下添加一个 paoding-dic-names.properties 属性文件，进行如下配置：

```
# 配置字典文件的读取编码
paoding.dic.charset=UTF-8
# 主要用于限制超过指定长度的中文字符串不进行分词
# 如果配置为零，则表示对任何长度的字符串都进行分词
paoding.dic.maxWordLen=0
# 字典文件名称以 "x-" 开头的会被忽略，不进行加载和编译
#paoding.dic.skip.prefix=x-
# 用于指定中日韩里单字形式的噪声字符（噪声字符即好比停用词，会被剔除掉）的字典文件名称
paoding.dic.noise-character=x-noise-character
# 用于指定中日韩里单字形式的噪声词语（噪声词语即好比停用词，会被剔除掉）的字典文件名称
paoding.dic.noise-word=x-noise-word
# 用于配置单位词汇的字典文件名称
paoding.dic.unit=x-unit
# 用于中文里的百家姓的字典文件名称
paoding.dic.confucian-family-name=x-confucian-family-name
# 用于定义字母和中文的连接词字典文件名称，比如：A 座、B 超、U 盘
paoding.dic.for-combinatorics=x-for-combinatorics
```

字典文件也正确放置之后，Paoding 分词器的运行环境就算配置好了，然后你就可以开始在 Lucene 中使用 Paoding 分词器啦。下面是在 Lucene 中使用 Paoding 分词器的简单示例：

```
Analyzer paodingAnalyzer = new PaodingAnalyzer();
String text = "奶茶妹妹抱女出镜小宝贝头戴粉色小发卡";
AnalyzerUtils.displayTokens(paodingAnalyzer, text);
```

其实 PaodingAnalyzer 还有一个重载的构造函数：

```
public PaodingAnalyzer(String propertiesPath, String mode, Knife knife) {
    this.propertiesPath = propertiesPath;
    this.mode = mode;
    this.knife = knife;
}
```

propertiesPath 参数用于指定 Paoding 分词器的核心属性文件 paoding-analysis.properties

的加载路径,如果用户未指定,那么默认值就是 `classpath:paoding-analysis.properties` 即默认会在 ClassPath 路径下去查找 `paoding-analysis.properties` 属性文件,如果你想要将其放置在其他路径下,你可以手动指定此参数。此参数有如下几种指定格式:

- `classpath:xxxxxx.properties` // 在 `classpath` 路径下查找 `xxxxxx.properties` 属性文件,当指定的属性文件不存在,会抛出异常;
- `ifexists:classpath:xxxxxx.properties` // 与上一种类似,唯一区别就是当指定的属性文件不存在,会直接跳过,不会抛出异常;
- `dic-home:xxxxxx.properties` // 在 `dic.home` 目录下查找 `xxxxxx.properties` 属性文件;
- `X://xxx/xxxx.properties` // 这种方式最好理解,直接指定为硬盘绝对路径;
- `xxx/xxxx.properties` // 如果指定为这种相对路径,此时是采用 `new File(path)` 方式加载,会根据实际运行不同而变化,如果程序允许在 Tomcat 下,那么就是相对于 Tomcat 的 `bin` 目录,如果程序是运行在普通的 Java Project,此时又是相对于当前项目的根目录,所以,不建议采用这种方式。

`mode` 参数表示分词模式,可选值有: `most-words`、`default`、`max-word-length`,不区分大小写。

`knife` 参数属于高级 API,用于提供给用户人工指定杀牛的“刀”,一般不用手动指定,用户也不需要了解 `knife` 的构建过程。因为在 `paoding-knives.properties` 属性文件中默认已经指定了 3 种 `knife` 的实现类,Paoding 分词器会自动构建这 3 个 `knife` 并装入放刀具的工具箱 (KnifeBox)。因此一般 `knife` 参数指定为 `null` 即可或者直接调用 `public PaodingAnalyzer(String propertiesPath,String mode)` 这个构造函数,从而忽略 `knife` 参数。

如果你想要在 Solr 中使用 Paoding 分词器,你需要自定义一个 `PaodingTokenzier Factory`,我提供的源码的 `solr` 包里已经创建了一个 `PaodingTokenzierFactory`,你只需要将这个类打包成 `jar`,然后将 `jar` 包导入 `core/lib` 目录下,然后你需要将源码里 `src/main/resources` 包下的 6 个 `properties` 属性文件全部复制到 `classpath` 路径下,假定你的 Solr 是部署在 Tomcat 下,那么此时 `classpath` 路径就是 `E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes`、`webapps\solr\WEB-INF\classes`,如图 4-6 所示。

然后你需要根据 `paoding-dic-home.properties` 属性文件里配置的 `paoding.dic.home` 参数来放置自定义字典文件,假定这里按照如下进行配置:

```
paoding.dic.home=classpath:dic
```

```
paoding.dic.detector.interval=60
```

因此,此时你需要在 `classpath` 路径下新建一个 `dic` 目录,然后将源码包里的 `dic` 目录下的所有文件全部复制到该 `dic` 目录下 (`.compiled` 目录可以不用复制),具体如图 4-7 和图 4-8 所示。

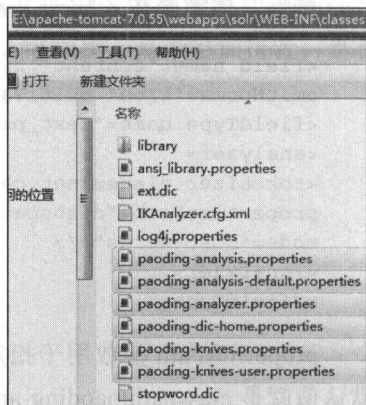


图 4-6 Paoding 属性文件放置路径





图 4-7 Paoding 分词器的 dic.home 目录

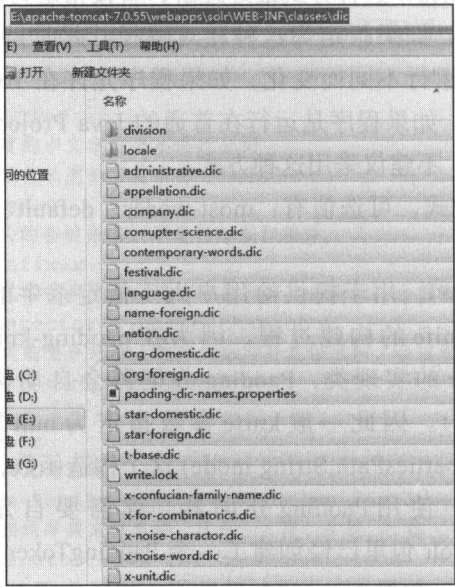


图 4-8 dic.home 目录下的自定义字典文件

最后，你需要在 schema.xml 中定义 Paoding 的 fieldType，配置示例如下所示：

```
<field name="content" type="text_paoding" indexed="true" stored="true"
omitNorms="true" multiValued="false"/>
<fieldType name="text_paoding" class="solr.TextField">
<analyzer>
<tokenizer class="net.paoding.analysis.solr.PaodingTokenizerFactory"
propertiesPath="classpath:paoding-analysis.properties"
mode="most-words" />
</analyzer>
</fieldType>
```

propertiesPath 参数用于指定 Paoding 分词器的核心属性文件的加载路径，如果不指定默认值就是 classpath:paoding-analysis.properties，这里仅仅只是为了演示此参数如何配置。

mode 参数用于指定分词模式，可选值：most-words、max-word-length、default，如果



不指定默认值就是 most-words。

到此，Paoding 分词器就能在 Solr 中正常工作了。不过经测试发现 Paoding 分词器对于具有分词歧义的字符串无法正确处理，比如：动漫游戏，Paoding 分词器会分成这样，如图 4-9 所示。

Field Value (Index)		Field Value	
动漫游戏			
Analyse Fieldname / FieldType: text_paoding			
PT	text	动漫	游戏
raw_bytes	[e5 8a a8 e6 bc ab]	[e6 bc ab e6 b8 b8]	[e6 b8 b8 e6 88 8f]
start	0	1	2
end	2	3	4
positionLength	1	1	1
type	word	word	word
position	2	5	9

图 4-9 Paoding 分词器对具有分词歧义的字符串的分词效果

不过，Paoding 分词器在某些方面确实更胜一筹，不管是代码设计思想还是配置文件和字典文件细粒度的分类划分管理机制，还是分词效率，都是一大亮点，美中不足是分词效果不太理想，而且可能会有 BUG，如果你决定选择使用 Paoding 分词器，那么就要做好随时填坑的风险。我在升级其源码使其支持 Lucene&Solr 5 的过程中已经填补了几个坑，断点 Debug 就是如果一整天的激情没有消失，那就鼓起勇气使用它吧。

### 4.3.5 Jcseg 分词器

Jcseg 是基于 mmseg 算法的一个轻量级开源中文分词器，同时集成了关键字提取，关键短语提取，关键句子提取和文章自动摘要等功能，并且提供了最新版本的 Lucene，Solr，Elasticsearch 的分词接口，Jcseg 自带了一个 jcseg.properties 文件用于快速配置而得到适合不同场合的分词应用，例如：最大匹配词长，是否开启中文人名识别，是否追加拼音，是否追加同义词等。

Jcseg 具有以下核心功能：

- 中文分词：mmseg 算法+Jcseg 独创的优化算法；
- 关键字提取：基于 textRank 算法；
- 关键短语提取：基于 textRank 算法；
- 关键句子提取：基于 textRank 算法；
- 文章自动摘要：基于 BM25+textRank 算法；
- 自动词性标注：目前只是基于词库，效果不是很理想；
- Restful api：嵌入 Jetty 提供了一个绝对高性能的 server 模块，包含全部功能的 http

接口，标准化 json 输出格式，方便各种语言客户端直接调用。

Jcseg 有以下 4 种分词模式：

- 简易模式：FMM 算法，适合速度要求场合；
- 复杂模式-MMSEG 四种过滤算法，具有较高的歧义去除，分词准确率达到 98.41%；
- 检测模式：只返回词库中已有的词条，很适合某些应用场合；
- 检索模式：细粒度切分，专为检索而生，除了中文处理外（不具备中文的人名，数字识别等智能功能）其他与复杂模式一致（英文，组词等）。

Jcseg 分词器具有以下亮点：

- 1) 支持自定义词库。在 `lexicon` 文件夹下，可以随便添加、删除、更改词库和词库内容，并且对词库进行了分类；
- 2) 支持词库多目录加载，配置 `lexicon.path` 中使用 ';' 隔开多个词库目录；
- 3) 词库分为简体、繁体、简繁体混合词库：可以专门适用于简体切分，繁体切分，简繁体混合切分，并且可以利用下面提到的同义词实现，简繁体的相互检索，Jcseg 同时提供了词库两个简单的词库管理工具来进行简繁体的转换和词库的合并；
- 4) 中英文同义词追加 / 同义词匹配 + 中文词条拼音追加，词库整合了《现代汉语词典》和 `cc-cedict` 辞典中的词条，并且依据 `cc-cedict` 词典为词条标上了拼音，依据《中华同义词词典》为词条标上了同义词（尚未完成）。更改 `jcseg.properties` 配置文档可以在分词的时候加入拼音和同义词到分词结果中；
- 5) 中文数字和中文分数识别，例如：“一百五十个人都来了，四十分之一的人。”中的“一百五十”和“四十分之一”。并且 Jcseg 会自动将其转换为阿拉伯数字加入到分词结果中。如：150, 1/40；
- 6) 支持中英混合词和英中混合词的识别（维护词库可以识别任何一种组合）。例如：B 超，x 射线，卡拉 ok，奇都 ktv，哆啦 a 梦；
- 7) 更好的英文支持，电子邮件，域名，小数，分数，百分数，字母和标点组合同（例如 C++, c#）的识别；
- 8) 自定义切分保留标点。例如：保留 &，就可以识别 k&r 这种复杂词条；
- 9) 复杂英文切分结果的二次切分：可以保留原组合，同时可以避免复杂切分带来的检索命中率下降的情况，例如 qq2013 会被切分成：qq2013/qq/2013, chenxin619315@gmail.com 会被切分成：chenxin619315@gmail.com/ chenxin/ 619315/gmail/com；
- 10) 支持阿拉伯数字 / 小数 / 中文数字基本单字单位的识别，例如 2012 年，1.75 米，38.6℃，五折，并且 Jcseg 会将其转换为“5 折”加入分词结果中；
- 11) 智能圆角半角，英文大小写转换；
- 12) 特殊字母识别：例如：Ⅰ，Ⅱ；特殊数字识别：例如：①，⑩；
- 13) 配对标点内容提取：例如：最好的 Java 书《java 编程思想》，‘畅想杯黑客技术大

赛’，被《、‘、“、『标点标记的内容（1.6.8 版开始支持）；

14) 智能中文人名/外文翻译人名识别。中文人名识别正确率达 94% 以上。（中文人名可以维护 lex-lname.lex, lex-dname-1.lex, lex-dname-2.lex 来提高准确率），（引入规则和词性后会达到 98% 以上的识别正确率）；

15) 自动中英文停止词过滤功能（需要在 jcseg.properties 中开启该选项，lex-stopwords.lex 为停止词词库）；

16) 词库更新自动加载功能，开启一个守护线程定时的检测词库的更新并且加载；

17) 自动词性标注（目前基于词库）。

Jcseg 的源码托管在 OSChina Git 上，下载地址为：<http://git.oschina.net/lionsoul/jcseg>。如果你使用的是 Jcseg 1.9.8 以上版本，你可以直接通过 Maven 引入 Jcseg 的 jar 包，pom.xml 中添加如下依赖即可：

```
<dependency>
<groupId>org.lionsoul</groupId>
<artifactId>jcseg-core</artifactId>
<version>1.9.9</version>
</dependency>
<dependency>
<groupId>org.lionsoul</groupId>
<artifactId>jcseg-analyzer</artifactId>
<version>1.9.9</version>
</dependency>
```

这里 Lucene 和 Solr 是共用一个 jcseg-analyzer-version.jar，其实 Solr 就一个 JcsegTokenizerFactory 类，完全可以单独打成一个 jar 包，这样仅仅在 Lucene 中使用就不需要额外添加 Solr 依赖了，这里 Jcseg 作者是打成一个 jar 包。

如果你没有 Maven 环境或者你也不想使用 Maven，那么你需要手动导入 jcseg-core 和 jcseg-analyzer 两个 jar 包。此时你需要导入源码到 IDEA 或 Eclipse 中，然后执行 Maven 的打包命令：mvn package -Dmaven.test.skip=true，然后就会自动在 Jcseg 各个模块的 target 目录下生成各模块的 jar 包。关于如何编译打包 Jcseg，这里以 IntelliJ-IDEA 为例，稍作演示，如图 4-10 和图 4-11 所示。

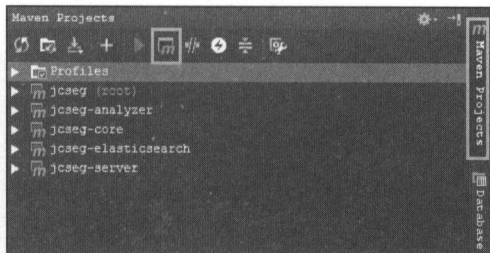


图 4-10 IDEA 中执行 Maven

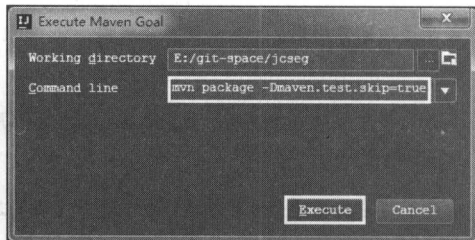


图 4-11 IDEA 中对 Jcseg 源码进行编译打包

依赖的 jar 包导入完成后，在 Lucene 中使用 Jcseg 似乎就变得很简单了，你只需要 `JcsegAnalyzer5X analyzer = new JcsegAnalyzer5X(JcsegTaskConfig.COMPLEX_MODE)`，即可完成 Jcseg 分词器的创建。但 JcsegAnalyzer5X 分词器对象在创建时，需要加载 Jcseg 需要的核心配置文件 `jcseg.properties`，`jcseg.properties` 属性文件是由 JcsegTaskConfig 类完成加载的，这里要分两种情况：`jcseg.properties` 属性文件的加载路径用户指定与未指定。如果用户指定了属性文件的加载路径，那么此时是采用 `new FileInputStream(path)` 方式加载，这种加载方式下，用户传入的 `path` 参数是硬盘绝对路径是最理想的情况，不建议采用相对路径。如果用户未指定属性文件的加载路径，那么此时 `jcseg.properties` 属性文件的加载逻辑如下：

- 1) 首先从 `jcseg-core-version.jar` 包内部的根目录下查找 `jcseg.properties`；
- 2) 如果没有找到，继续从 Classpath 路径下查找 `jcseg.properties`；
- 3) 如果还是没有找到，继续从 `System.getProperty("user.home")` 目录下查找 `jcseg.properties`；这里的 `System.getProperty("user.home")` 表示当前系统的登录用户的家目录（不同操作系统的家目录不一致，具体值自己运行打印 `System.getProperty("user.home")` 便知）。

因此，我们可以了解到，`jcseg.properties` 属性文件可以不用指定，因为它默认已经打包到 `jcseg-core-version.jar` 包内。但我们一般需要修改默认的配置，所以 `jcseg.properties` 属性文件最好还是需要配置的，最简单的方式就是复制一份，然后放置到 classpath 下。下面是一份 `jcseg.properties` 属性文件的配置示例：

```
#####Jcseg 核心功能相关配置参数
# 匹配的 Token 文本最大长度
jcseg.maxlen = 7

# 是否自动识别中文人名，1 表示启用
jcseg.icnname = 1

# 英文 + 中文的混合文本中，中文字符的最大个数
jcseg.mixcnlen = 3

# 一对标点符号之间的文本最大长度
jcseg.pptmaxlen = 7

# 中文姓氏修饰词的最大长度，比如：“老陈”、“小陈”中的老、小
jcseg.cnmaxlnadron = 1

# 是否剔除停用词，1 表示剔除停用词
jcseg.clearstopword = 0

# 是否将中文数字转换成阿拉伯数字，1 表示启用
jcseg.cnumtoarabic = 1

# 是否将中文里的分数转换成阿拉伯里的分数，1 表示启用
jcseg.cnfratoarabic = 0

# 分词器不认识的词是否保留，1 表示保留，0 表示剔除
```

```

jcseg.keepunregword = 1

# 对于复杂的英文单词是否启用二次切分, 1 表示启用
jcseg.ensencondseg = 1

# 次要简单 Token 的最小长度, 建议设置为大于 1
jcseg.stokenminlen = 2

# 中文人名识别功能需要用到的阈值
jcseg.nsthreshold = 1000000

# 定义需要保留的标点符号, 否则标点符号会被剔除掉
jcseg.kepppunctuations = @#%.&+

#### 字典文件相关配置参数
# 字典文件名称的前缀
lexicon.prefix = lex

# 字典文件名称的后缀
lexicon.suffix = lex

# 字典文件的绝对路径
# 自 Jcseg1.9.2 版本开始, 支持配置多个路径, 使用分号 ; 分割多个路径
# 例如:
#lexicon.path = /home/chenxin/lex1;/home/chenxin/lex2 (Linux)
#lexicon.path = D:/jcseg/lexicon/1;D:/jcseg/lexicon/2 (Windows)
#lexicon.path=/Code/java/JavaSE/jcseg/lexicon
#lexicon.path = {jar.dir}/lexicon
#{jar.dir} 表示相对于 jcseg-core-{version}.jar 包内部的根目录
# 自 Jcseg1.9.9 版本开始, Jcseg 默认从 classpath 路径下加载字典文件
lexicon.path = E:/apache-tomcat-7.0.55/webapps/solr/WEB-INF/classes/lexicon

# 是否自动检测字典文件更新并自动重新加载, 1 表示开启, 0 表示禁用
lexicon.autoload = 0

# 自动加载字典文件的轮询周期, 即每间隔多久检测字典文件是否已更新, 单位: 秒
lexicon.polltime = 300

#### 字典中词汇加载内容相关配置参数
# 是否加载词语的词性, 1 表示加载
jcseg.loadpos = 1

# 是否加载拼音, 1 表示加载
jcseg.loadpinyin = 0

# 是否加载同义词, 1 表示加载
jcseg.loadsyn = 1

```

需要着重关注的是 `lexicon.path` 这个配置项，用于配置字典文件的加载路径。Jcseg 的所有字典文件是由 Jcseg-core 模块里的 `ADictionary` 类的 `loadWords` 方法负责加载。当创建 `JcsegAnalyzer5X` 对象时，最终都会调用这个构造函数。

```
public JcsegAnalyzer5X(int mode, JcsegTaskConfig config) {
    this(mode, config, DictionaryFactory.createSingletonDictionary(config));
}
```

`DictionaryFactory` 字典工厂类的 `createSingletonDictionary` 方法在构造单例的 `Dictionary` 时，最终会调用 `ADictionary.createDefaultDictionary(JcsegTaskConfig config, boolean sync, boolean loadDic)` 函数，此函数内部会读取用户在 `jcseg.properties` 属性文件中配置的 `lexicon.path` 参数值，如果用户没有指定 `lexicon.path` 参数值或者 `lexicon.path` 参数值配置为 `null`，那么会调用 `loadClassPath` 方法完成所有字典文件的加载，如果用户配置了 `lexicon.path` 参数值且不为 `null`，那么会调用 `loadDirectory` 方法完成所有字典文件的加载。先来观摩下 `loadDirectory` 方法，大致逻辑就是通过 `File` 类的 `listFiles` 方法遍历当前目录下的所有文件，并通过用户在 `jcseg.properties` 属性文件中配置的 `lexicon.prefix` 和 `lexicon.suffix` 两个参数值来过滤掉字典文件名称不符合要求的文件，然后循环通过 `load(file)` 方法加载并解析每个符合要求的字典文件。再来看看 `loadClassPath` 是如何加载字典文件的，源码如下所示：

```
public void loadClassPath() throws IOException{
    String __suffix = config.getLexiconFileSuffix();
    String __prefix = config.getLexiconFilePrefix();
    Class<?> dClass = this.getClass();
    CodeSource codeSrc =
this.getClass().getProtectionDomain().getCodeSource();
    if ( codeSrc == null ) {
        return;
    }
    // 关键点 1
    String codePath = codeSrc.getLocation().getPath();
    if ( codePath.toLowerCase().endsWith(".jar") ) {
        ZipInputStream zip = new
ZipInputStream(codeSrc.getLocation().openStream());
        while (true) {
            ZipEntry e = zip.getNextEntry();
            if ( e == null ) {
                break;
            }
            String fileName = e.getName();
            if ( fileName.startsWith("lexicon/"+__prefix)
&& fileName.endsWith(__suffix) ) {
                load(dClass.getResourceAsStream("/"+fileName));
            }
        }
    } else {
        // 关键点 2
```



```
// 此时 class 字节码文件一般在当前项目的 bin 或 out 目录下 (即源码编译输出目录)
// 若是 Maven 项目, 那么源码编译输出目录一般默认是 target/classes. 而若是 Web
// Project, 则源码编译输出目录是 WEB-INF\classes 目录下,
// 即常说的 classpath 路径下
loadDirectory(codePath+"/lexicon");
}
}
```

关键点 1: 获取当前类的 class 字节码文件所在根目录, 如果以 .jar 结尾, 则表明当前类的 class 字节码文件在 jar 包内部, 此时需要通过 ZipInputStream 类读取 jar 包, 在 jar 包内部查找 lexicon 目录, 并根据 prefix 和 suffix 参数加载符合要求的字典文件。而默认字典文件已经打包在 jcseg-core-version.jar 内部, 即你可以不配置提供字典文件, 但一般我们需要根据自己的需求对字典文件进行频繁添加修改, 所以建议还是在 lexicon 目录下配置自定义一份字典文件。

关键点 2: 如果当前类的 class 字节码文件不在 jar 包内部, 那么也就一定在源码编译输出目录中, 由于 Solr 自身是个 Web Project, 因此源码编译输出目录即 WEB-INF\classes。确定了字典文件的目录位置, 剩下的字典文件加载逻辑就与用户手动指定了字典文件加载路径的情况相同, 都是 loadDirectory(String lexDir) 方法完成加载。

如果了解了 jcseg.properties 属性文件如何配置以及字典文件根目录 lexicon.path 如何配置, 那么在 Lucene 中使用 Jcseg 分词器的障碍就全部扫清了, 你可以开始愉快的使用 Jcseg 分词器啦。

如果你想要在 Solr 中使用 Jcseg 分词器, 那么你需要导入 jcseg-core-version.jar 和 jcseg-analyzer-version.jar 这两个 jar 包。由于 Solr 需要的 JcsegTokenizerFactory 类已经包含在 jcseg-analyzer-version.jar 包内部, 所以你不需要额外导入其他 jar 包, 但我还是建议你 JcsegTokenizerFactory 类单独打成一个 jar 包。依赖的 jar 包需要放置在当前 core/lib 目录下。然后你需要从源码包中获取一份 jcseg.properties 属性文件, 然后将其复制到 classpath 路径下, 即 Solr 自身项目的 WEB-INF\classes 目录, 比如: E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes, 然后你需要对 jcseg.properties 属性文件稍作配置, 这里我们只配置 lexicon.path 参数, 其他配置项你们根据注释酌情修改, 参考配置示例如下所示:

```
lexicon.path=E:/apache-tomcat-7.0.55/webapps/solr/WEB-INF/classes/lexicon
```

注意, 这里的路径最好是使用反斜杠字符 \, 不要使用正斜杠 /, 尽量与 Linux 环境下的风格统一, 仅仅只是个建议。然后你需要根据 lexicon.path 配置, 在 WEB-INF\classes\ 目录下新建 lexicon 目录, 然后从源码包中的 vendors\lexicon 目录下获取所有字典文件, 并将其全部复制到我们刚刚在 WEB-INF\classes\ 目录下新建的 lexicon 目录。

最后你需要在当前 Core 的 schema.xml 中配置 <fileType> 元素, 下面是一个简单的配置示例:

```
<field name="content" type="text_pading" indexed="true" stored="true"
```



```
omitNorms="true" multiValued="false"/>
<fieldtype name="text_jcseg" class="solr.TextField">
<analyzer>
<tokenizer
class="org.lionsoul.jcseg.analyzer.v5x.JcsegTokenizerFactory"
mode="complex"/>
</analyzer>
</fieldtype>
```

其中 mode 表示分词模式，内置了 4 种分词模式，分别是：复杂模式、简易模式、检测模式、最多模式，分别对应的参数值是：complex、simple、detect、search。如果 mode 参数未指定，那么默认值就是 complex。

目前，我们的 jcseg.properties 属性文件是直接放置在 classpath 路径下，而 jcseg 对 jcseg.properties 属性文件的加载逻辑默认会搜寻 classpath 路径，从而能找到必需的 jcseg.properties 属性文件。但有时，你可能希望手动指定 jcseg.properties 属性文件的加载路径，在 Lucene 中，这很好解决，你只需要调用如下构造函数即可实现：

```
public JcsegAnalyzer5X(int mode, String proFile) {
this(mode, new JcsegTaskConfig(proFile));
}
```

其中 proFile 参数即用于提供给用户指定 jcseg.properties 属性文件的加载路径。然而，在 Solr 中，你只能通过 schema.xml 来配置，但 Jcseg 作者提供的 JcsegTokenizerFactory 类并没有提供用于指定 jcseg.properties 属性文件的加载路径的配置参数，因此我们需要自己修改 JcsegTokenizerFactory 类的源码。首先在 JcsegTokenizer 类里添加一个带有 jcseg.properties 属性文件的加载路径的构造参数，示例代码如下所示：

```
public JcsegTokenizer(String propertiesPath,int mode) throws JcsegException,
IOException {
JcsegTaskConfig config = new JcsegTaskConfig(propertiesPath);
ADictionary dic = DictionaryFactory.createSingletonDictionary(config);
segmentor = SegmentFactory.createJcseg(mode, new Object[]{config, dic});
segmentor.reset(input);
}
```

JcsegTokenizerFactory 类里添加了一个 propertiesPath 属性用于接收用户传入的 jcseg.properties 属性文件加载路径参数。具体实现代码如下所示：

```
/**
 * Created by Lanxiaowei
 */
public class JcsegTokenizerFactory extends TokenizerFactory
{
    /** 分词模式: complex、simple、detect、search*/
    private int mode;
    /**jcseg.properties 属性文件的加载路径 */
```

```

private String propertiesPath;
/**Jcseg 配置对象*/
private JcsegTaskConfig config;
/**Jcseg 字典对象*/
private ADictionary dic;

public JcsegTokenizerFactory(Map<String, String> args) {
    super(args);
    String _mode = args.get("mode");
    if (_mode == null || "".equals(_mode)) {
        this.mode = JcsegTaskConfig.COMPLEX_MODE;
    } else {
        _mode = _mode.toLowerCase();
        if ("simple".equalsIgnoreCase(_mode)) {
            mode = JcsegTaskConfig.SIMPLE_MODE;
        } else if ("detect".equalsIgnoreCase(_mode)) {
            mode = JcsegTaskConfig.DETECT_MODE;
        } else if ("search".equalsIgnoreCase(_mode)) {
            mode = JcsegTaskConfig.SEARCH_MODE;
        } else {
            mode = JcsegTaskConfig.COMPLEX_MODE;
        }
    }
    String propertiesFilePath = args.get("propertiesFilePath");
    if (null == propertiesFilePath || "".equals(propertiesFilePath)) {
        config = new JcsegTaskConfig(true);
        dic = DictionaryFactory.createSingletonDictionary(config);
    } else {
        this.propertiesPath = propertiesFilePath;
    }
}

@Override
public Tokenizer create(AttributeFactory factory) {
    try {
        if (null != config && null != dic) {
            return new JcsegTokenizer(mode, config, dic);
        }
        return new JcsegTokenizer(this.propertiesPath, this.mode);
    } catch (JcsegException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

如下修改完成之后，你需要将他们重新打包成 jar 包并导入，然后你就在 schema.xml

的中应用我们自定义的 `propertiesFilePath` 参数来指定 `jcseg.properties` 属性的加载路径了。具体配置示例如下所示：

```
<fieldtype name="text_jcseg" class="solr.TextField">
<analyzer>
<tokenizer
class="org.lionsoul.jcseg.analyzer.v5x.JcsegTokenizerFactory"
mode="complex"
propertiesFilePath="E:/jcseg/conf/jcseg.properties"/>
</analyzer>
</fieldtype>
```

关于 `JcsegTokenizer` 和 `JcsegTokenizerFactory` 这两个类修改后的详细源代码请访问我的 Github 获取，访问地址：<https://github.com/yida-lxw/jcseg>。


如果某些词语 Jcseg 不认识分不出来，那么此时你需要在字典文件里添加新词，Jcseg 对字典文件分类很细致，比如民族相关词语是定义在 `lex-cn-mz.lex` 字典文件中，节日相关词语是定义在 `lex-festival.lex` 字典文件中，网络流行词语是定义在 `lex-net.lex` 字典文件中，其他字典文件的作用请自行打开文件查阅便知。当然也可以自己往字典目录里添加新的字典文件，你只需要复制任意一份字典文件并改名，但字典文件名称必须符合默认 `jcseg.properties` 属性文件中 `lexicon.prefix` 和 `lexicon.suffix` 两个参数限制的规则即字典文件名称必须以 `lex` 开头并后缀名为 `dic`。当然你可以删除 `lexicon.prefix` 和 `lexicon.suffix` 两个参数，取消其对于字典文件名称的限制。然后可以打开字典文件进行编辑，添加任意你想要被 Jcseg 分词器正确切分的词语。Jcseg 中的字典文件内容定义格式如下：

```
柯基犬 /nr/ke ji quan/ 柯基
么么哒 /n/me me da/null
```

依次表示：词语、词性、汉语拼音、同义词。如果某一项为空，你可以填上 `null`。注意，每一行只能是这 4 项，每一项使用反斜杠字符分割，如果每一行的项数不等于 4，那么这行配置就无效。注意，如果你想要添加汉语拼音功能，那么你需要在 `jcseg.properties` 属性文件中将 `jcseg.loadpinyin` 设置为 1。同理，如果你想要添加同义词功能，那么你需要在 `jcseg.properties` 属性文件中将 `jcseg.loadsyn` 设置为 1。因为每一行只能是 4 项且第 4 项为同义词，这也意味着每一行你只能定义一个同义词，假如一个词语它有  $N(N > 1)$  个同义词，此时该如何为一个词语定义多个同义词呢？你可以选择为同一个词语定义多行，只要每行的第 4 项保持不同即可实现，具体示例如下所示：

```
姑娘 /n/gu liang/ 菇凉
姑娘 /n/gu liang/ 姑凉
```

上面配置表示“菇凉”和“姑凉”是“姑娘”的同义词。如果你想要使用停用词功能，首先你需要开启 `jcseg.properties` 属性文件中的 `jcseg.clearstopword` 配置项并将其设置为 1。然后你需要维护 `lex-stopword.lex` 字典文件，把你想要剔除掉的词语添加进此字典文件里即可。

 **注意** 修改了字典文件后，如果你没有在 `jcseg.properties` 属性文件中配置字典文件自动更新检测的话，那么你需要重新加载你的 `core`，或者你直接设置 `lexicon.autoload = 1` 即开启字典文件自动更新检测功能。

此外 Jcseg 的 `server` 模块还提供了一个基于嵌入式的 Jetty 容器的分词服务，支持 Restful 风格的分词接口，即用户 `post` 一个需要分词的文本到 Jcseg Server，然后 Jcseg Server 会以 `Json` 格式返回 Server 端分词后提取出来的关键字给调用客户端。想要使用 Jcseg Server 模块提供的功能，你首先需要编译打包 Jcseg-server 模块，然后得到一个 `jcseg-server-version.jar`，然后执行命令：

```
java -jar jcseg-server-{version}.jar ./jcseg-server.properties
```

这样就可以启动 Jcseg Server 了。Jcseg Server 启动时需要加载的 `jcseg-server.properties` 属性文件可以从 Jcseg 源码包中获取到。启动 Jcseg Server 之前，你需要事先配置好 Jcseg Server。Jcseg Server 相关配置需要在 `jcseg-server.properties` 属性文件中定义。`jcseg-server.properties` 配置详解如下所示：

# Jcseg Server 的配置文件采用标准的 JSON 数据格式进行定义

```
{
  # jcseg server 相关配置参数
  "server_config": {
    # Jcseg server 监听端口号
    "port": 1990,
    # 默认 HTTP 通信的字符编码
    "charset": "utf-8",
    # Http 连接空闲超时时间，单位：毫秒
    "http_connection_idle_timeout": 60000,
    # Jetty 线程池的最大线程数
    "max_thread_pool_size": 200,
    # Jetty 线程池中的线程空闲超时时间，单位：毫秒
    "thread_idle_timeout": 30000,
    # Http 输出缓冲区大小，单位：byte
    "http_output_buffer_size": 32768,
    # Http Request 请求头大小
    "http_request_header_size": 8192,
    # Http Response 响应头大小
    "http_response_header_size": 8192
  },
```

# Jcseg 通用配置，具体请参看 `jcseg.properties` 属性文件里的注释

```
"jcseg_global_config": {
  "jcseg_maxlen": 7,
  "jcseg_icnname": true,
  "jcseg_mixcnlen": 3,
  "jcseg_pptmaxlen": 7,
  "jcseg_cnmaxlnadron": 1,
```

```

    "jcseg_clearstopword": false,
    "jcseg_cnnumtoarabic": true,
    "jcseg_cnfratoarabic": false,
    "jcseg_keepunregword": true,
    "jcseg_ensencondseg": true,
    "jcseg_stokenminlen": 2,
    "jcseg_nsthreshold": 1000000,
    "jcseg_keeppunctuations": "@#%.&+"
  },
  # Jcseg 字典相关配置, 具体请参看 jcseg.properties 属性文件里的注释
  "jcseg_dict": {
    "master": {
      # path 设置为 null, 那么将会从 classpath 路径下加载字典文件
      path: null,
      # "path": [
      #   "{jar.dir}/lexicon"
      #   # absolute path here
      #   # "/java/JavaSE/jcseg/lexicon"
      # ],
      "loadpos": true,
    },
    "loadpinyin": true,
    "loadsyn": true,
    "autoload": true,
    "polltime": 300
  }

  # 更多自定义参数
  # , "name" : {
  #   "path": [
  #     "absolute jcseg standard lexicon path 1",
  #     "absolute jcseg standard lexicon path 2"
  #     ...
  #   ],
  #   "autoload": 0,
  #   "polltime": 300
  # }

},
# JcsegTaskConfig 实例相关配置参数
# @注意:
# JcsegTaskConfig 实例相关配置参数默认全部继承自 Jcseg 的通用配置
"jcseg_config": {
  "master": {
    # 继承并覆盖 Jcseg 的通用配置
    "jcseg_pptmaxlen": 0,
    "jcseg_cnfratoarabic": true,
    "jcseg_keepunregword": false
  }
  # 这里的配置参数用于关键字、关键短语、关键句、摘要提取
  # @注意: 如果你需要 Jcseg Server 为你提供关键字、关键短语、关键句、摘要提取等功能服务,
  # 那么请不要删除 extractor 配置
  , "extractor": {

```

```

"jcseg_pptmaxlen": 0,
"jcseg_clearstopword": true,
"jcseg_cnumtoarabic": false,
"jcseg_cnfratoarabic": false,
"jcseg_keepunregword": false,
"jcseg_enscondseg": false
}
# 自定义配置参数
# , "name": {
#   ...
# }
},
# Jcseg tokenizer 实例相关配置参数
# 你可以通过访问如下链接来获取实例服务:
# http://jcseg_server_host:port/tokenizer/instance_name
# URL 中的 instance_name 表示你的实例名称
"jcseg_tokenizer": {
  "master": {
    # jcseg tokenizer 算法:
    # 1: SIMPLE_MODE
    # 2: COMPLEX_MODE
    # 3: DETECT_MODE
    # 详情请参阅 org.lionsoul.jcseg.tokenizer.core.JcsegTaskConfig 类的源码
    "algorithm": 2,

    # 字典实例名称
    "dict": "master",

    # JcsegTaskConfig 实例名称
    "config": "master"
  }
  # extractor 实例
  , "extractor": {
    "algorithm": 2,
    "dict": "master",
    "config": "extractor"
  }
  # 自定义配置参数
  # , "name": {
  #   ...
  # }
}

```

其实上面所有配置你保持默认即可正确启动 Jcseg Server, 请根据实际需求再修改 Jcseg Server 的配置项, 比如你想要修改字典文件, 那么你就需要修改 Jcseg 字典相关配置即 jcseg\_dict 部分。各项配置项的详细含义请自己对照上面的注释去理解, 有关 Jcseg Server 提供的 Restful API 相关内容, 请访问链接 <http://git.oschina.net/lionsoul/jcseg> 作更详细的了解。

### 4.3.6 Ictclas 分词器

Ictclas 分词器是中科院张华平博士耗时 1 年基于多层隐码模型的汉语词法分析系统使用 C++ 编写的一款中文分词器，现已更名为 NLPIR。NLPIR 分词系统主要功能包括中文分词、词性标注、命名实体识别、用户词典功能、支持 GBK 编码、UTF8 编码、BIG5 编码、新增微博分词以及新词发现与关键词提取。国内很多中文分词器都参考 ictclas 的算法，比如免费的 Ansj 中文分词器。然而它并不是免费的，而且在 Java 环境中使用比较麻烦。

Ictclas 官方地址：<http://ictclas.nlpir.org/>，你可以从官方下载获取到它的下载包，下载包中包含了在 Java 中调用 ictclas 分词器的示例，本质就是借助 JNA 调用 DLL 动态链接库。为了方便大家在 Java 中使用 Ictclas，我在官方提供的 demo 基础上，编写了一套基于 Lucene&Solr 5.x API 的接口，这样大家就能直接以 new IctclasAnalyzer(); 方式来使用它，详细代码请访问我的 github 获取。下面是在 Lucene 中使用 Ictclas 分词器的简单示例：

```
public class IctclasAnalyzerTest {
    public static void main(String[] args) throws IOException {
        // 待分词的文本
        String text = "《大话3》被吐槽：情怀很珍贵请勿滥消费";
        // 停用词典文件加载路径
        String stopwordPath = "E:/apache-tomcat-7.0.55/webapps/solr/WEB-INF/classes/stopword.dic";
        // 用户自定义扩展字典文件加载路径，可以是绝对路径
        String userDicPath = "E:\\git-space\\ictclas-2016\\dict\\user_dict.txt";
        Analyzer analyzer = new IctclasAnalyzer(false, stopwordPath, userDicPath);
        displayTokens(analyzer, text);
    }
}
```

首先你需要在项目的 classpath 下添加一个 ictclas.properties 属性文件，下面该属性文件的配置示例：

```
#dll_path=win64/NLPIR.dll
dll_path=D:/ictclas/dll/NLPIR.dll
data_path=D:/ictclas/
stopword_path=E:/apache-tomcat-7.0.55/webapps/solr/WEB-INF/classes/stopword.dic
userdic_path=dict/user_dict.txt
add_speech=true
```

**dll\_path**：用于配置 Ictclas 分词器初始化时需要加载的 NLPIR.dll 文件的路径，如果你配置为相对路径，比如 `dll_path= win64/NLPIR.dll`，那么此时就是在当前项目的根目录下查找一个 win64 文件夹，在该文件夹下查找 NLPIR.dll 并 load 它。NLPIR.dll 与操作系统有关，Linux 系统环境下需要加载 NLPIR.so，由于系统又分 32bit 和 64bit，因此 NLPIR 文件分 4 种类型。dll\_path 配置项并不是必须指定的，当用户未指定 dll\_path，程序会自动根据当前操作系统环境去当前项目根目录下查找 NLPIR 文件。比如 window64 系统环境下会查找 win64/NLPIR.dll，linux32 系统环境下会查找 linux32/NLPIR.so，其他系统同理。dll\_path



配置项必须配置正确, 否则 Ictclas 会初始化失败。

**data\_path**: 用于配置 Ictclas 分词器的数据目录, 该目录下存放了很多分词器自定义数据格式的字典文件, 大部分文件并不能直接打开查看。不配置此参数默认会在当前项目的根目录下查找 Data 目录。**data\_path** 配置项必须配置正确, 否则 Ictclas 会初始化失败。

**stopword\_path**: 用于配置停用词字典文件的加载路径, 你可以将停用词字典文件放置在 classpath 路径下, 或者直接指定为硬盘绝对路径。

**userdic\_path**: 用于指定用户扩展字典文件的加载路径, 你可以指定为硬盘绝对路径, 如果指定为相对路径, 此时就是相对当前项目的根目录。

**add\_speech**: 配置项用于配置是否为词语添加词性标注。上述这些参数都不是必须指定的, 除非你很清楚不配置时它们的默认值情况, 否则请显示指定他们。

为了在 Solr 中使用 Ictclas, 为此我编写了 IctclasTokenizerFactory, 你需要编译源码并打包成 jar 并导入 Solr, 然后将 ictclas.properties 属性文件复制到 classpath 环境下, 并根据 ictclas.properties 中的配置将 NLPIR 文件、停用词字典、用户扩展字典等文件放置在正确位置, 然后你需要在 Solr 的 schema.xml 中配置 <fieldType>, 配置实例如下所示:

```
<fieldtype name="text_ictclas" class="solr.TextField">
<analyzer>
<tokenizer class="com.ictclas.solr.IctclasTokenizerFactory"
dllPath="D:/ictclas/dll/NLPIR.dll" stopwords="D:/ictclas/dic/stopword.dic"
userDic="D:/ictclas/dic/user_dic.txt" addSpeech="true"/>
</analyzer>
</fieldtype>
```

由于 NLPIR 并不免费, 如果你不付费购买授权, 那么你能只能用作学习研究, 并不能将其用于自己项目中, 而且它没有提供 Java 接口, 只能通过加载 DLL 或 SO 文件来间接调用 C/C++ 接口。使用过程中出现问题调试定位问题麻烦。我编写的这套为了兼容 Lucene&SolrAPI 的示例代码仅供大家学习参考。

#### 4.3.7 FudanNLP

FudanNLP 是复旦大学使用 Java 语言开发的基于 LGPL 3.0 许可证开源的一款中文分词器, 它主要包括以下主要功能:

##### □ 中文处理

- 中文分词、词性标注、实体名识别、句法分析、时间表达式识别

##### □ 信息检索

- 文本分类、新闻聚类
- 兼容 Lucene 4.x

##### □ 机器学习

FudanNLP 源代码托管在 Google code 上, 访问地址为: <http://code.google.com/p/fudannlp/>,

不过你需要翻墙才能访问下载到源码包。目前我已经将其改成基于 Maven 构建并将 Lucene 依赖提升到 5.x, 添加了核心配置文件 FudanNLP.xml 使其实现可配置化、添加了停用词字典和自定义扩展字典文件, 同时添加了对 Solr 5.x 的支持以及修复了一些 BUG。如果你感兴趣, 可以访问我的 Github 获取我改造后的 FudanNLP 源码进行学习: <https://github.com/yida-lxw/FudanNLP-1.6.1>。

想要在 Lucene 中使用 FudanNLP 分词器, 首先你需要获取它的源码, 然后编辑打包得到一个 jar, 并将其导入项目的 classpath 路径下。然后你需要创建一个 FudanNLPAnalyzer 对象实例, 具体请参阅 FudanNLPAnalyzer 类的构造函数, 如下:

```
/**
 * FNLPAAnalyzer 构造函数
 * @param modePath      模型文件目录, 可以是硬盘绝对路径, 也可以是 classpath 下相对路径
 * @param mode          模型类型, 可选值有: seg, tag, seg_tag, ner, parser, all
 * @param userDicPath   用户自定义扩展字典文件或目录路径
 * @param stopwordPath  用户扩展停用词字典文件路径
 * @param usePOSFilter   是否启用词性过滤器
 * @param ambiguity     是否启用对用户自定义扩展字典里的词语进行模糊处理
 */
public FudanNLPAnalyzer(String modePath, String mode, String userDicPath,
                        String stopwordPath, boolean usePOSFilter, boolean ambiguity)
```

各构造参数的含义具体请看代码注释, 这些构造参数可以全部不指定, modePath 如果未指定, 那么默认会在当前项目的 classpath 路径查找一个 models 目录, 你需要将模型文件放置在该 models 目录下。mode 参数表示模型文件的类型, 内部会根据这个不同的类型值加载 models 目录下相应的以 .m 结尾的模型文件, 从而实现不同分词组件的构建和功能组合, 比如 seg 就是基本的中英文分词组件, tag 就是基本的词性标注分词组件, seg\_tag 就是中英文分词与词性标注的组合。usePOSFilter 表示是否添加词性过滤器即词性不符合要求的词语将会被当作停用词而剔除掉, 默认值为 true。ambiguity 参数表示是否对自定义扩展字典里的词语进行模糊处理, 具体这个参数的含义和作用请查阅 ChineseWordSegmentation 类并运行其中的测试方法自行去感受下。ambiguity 参数默认值为 true 即开启对词语的模糊处理。当然, FudanNLPAnalyzer 构造函数里的 6 个参数都可以通过 FudanNLP.xml 来进行配置, 你只需要在 classpath 路径下添加一个 FudanNLP.xml 配置文件, 配置样例如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>FudanNLP Analyzer 配置</comment>
<!-- 用户可以在这里配置自己的自定义扩展字典, 可以是字典文件路径,
也可以是字典文件所在目录路径, 多个路径之间使用分号 ; 分割 -->
<entry key="user_dic">ext.dic;</entry>
<!-- 用户可以在这里配置自己的停用词字典, 可以是字典文件路径,
也可以是字典文件所在目录路径, 多个路径之间使用分号 ; 分割 -->
<entry key="stop_word_dic">stopword.dic;</entry>
```

```

<!-- 用户可以在这里配置模型文件所在目录路径 -->
<entry key="mode_path">models</entry>
<!-- 用户可以在这里配置模型文件的类型，可选值有：seg,tag,seg_tag,ner,parser,all-->
<entry key="mode">seg</entry>
<!-- 用户可以在这里配置是否启用 POSFilter 过滤器 :true/false-->
<entry key="use_pos_filter">true</entry>
<!-- 用户可以在这里配置是否启用对用户自定义扩展字典里的词语进行模糊处理 :true/false-->
<entry key="dic_ambiguity">true</entry>
</properties>

```

如果用户构造函数里指定了这些参数，同时 FudanNLP.xml 里也指定了，那么此时以构造函数里的参数值为准。

在测试代码包 org.fnlp.app.lucene.test 下，有几个测试示例，演示了如下在 Lucene 5.x 下使用 FudanNLP 分词器创建索引以及完成索引搜索，建议大家单步跟踪下代码执行流程，这样有利于你更好的理解和使用 FudanNLP 分词器。

如果你想要在 Solr 中使用 FudanNLP 分词器，那么你首先需要将 org.fnlp.app.solr 包下的 3 个类单独打包成 jar，然后将整个源码包（除开 Solr 相关的 3 个类）打成一个 jar，然后将这 2 个 jar 包复制到 core/lib 目录下，之后你需要将 FudanNLP.xml 核心配置文件以及模型文件目录 models 全部复制到 classpath 路径下即 E:\apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes（这里假定你的 Solr 是部署在 Tomcat webapps 目录下）。然后你需要按照你实际的需求配置 FudanNLP.xml，比如停用词字典文件和自定义扩展字典文件的加载路径，配置完成后需要根据配置的路径将字典文件复制到对应位置。这里建议将字典文件的路径配置成硬盘绝对路径，除非你很清楚字典文件背后的加载机制。具体字典文件以及核心配置文件 FudanNLP.xml 背后的加载机制请参阅 edu.fudan.util.FileUtils 工具类里的 public static File makeFile (String filePath) 方法实现。最后需要在 schema.xml 中配置 FudanNLP 分词器的 fieldType，具体配置示例如下：

```

<fieldtype name="text_fudannlp" class="solr.TextField">
<analyzer>
<tokenizer class="org.fnlp.app.solr.SentenceTokenizerFactory"
modePath="models" mode="seg" ambiguity="true"
userDicPath="D:/ictclas/dic/user_dic.txt" />
<filter class="org.fnlp.app.solr..WordTokenFilterFactory"
stopwords="stopword.dic" />
<filter class="org.fnlp.app.solr..POSTaggingFilterFactory"
enable_pos_increment="true" />
</analyzer>
</fieldtype>

```

<tokenizer> 和 <filter> 元素里配置的属性值参数同样可以通过 FudanNLP.xml 来进行配置，你只需要在 classpath 路径下放置一个 FudanNLP.xml 并对其进行配置即可。但在 schema.xml 中配置的参数优先，除非用户未指定该属性值参数，才会以 FudanNLP.xml 配置文件中的配置为准。

### 4.3.8 HanLP

HanLP 是由一系列模型与算法组成的 Java 开源工具包，目标是普及自然语言处理在生产环境中的应用。HanLP 具备功能完善、性能高效、架构清晰、语料时新和可自定义的特点。

HanLP 具备以下功能：

- ☐ 中文分词：最短路分词、N-最短路分词、CRF 分词；
- ☐ 支持用户自定义扩展字典；
- ☐ 支持词性标注；
- ☐ 支持中英文人名、英文译名、日本人名、地名、机构名自动识别；
- ☐ 支持关键词、短语、摘要提取，基于 Google 的 TextRank 算法实现；
- ☐ 支持拼音转换：包括声母韵母音调，支持多音字的拼音转换；
- ☐ 支持对中文繁体字分词，支持简繁歧义词；
- ☐ 支持文本推荐：语义推荐、拼音推荐、字词推荐；
- ☐ 支持依存句法分析：基于神经网络的高性能依存句法分析器、MaxEnt 依存句法分析、CRF 依存句法分析；
- ☐ 提供语料库工具：包括分词语料预处理、词频词性词典制作、BiGram 统计、词共现统计。CoNLL 语料预处理、CoNLL UA/LA/DA 评测工具。

在提供丰富功能的同时，HanLP 内部模块坚持低耦合、模型坚持惰性加载、服务坚持静态提供、词典坚持明文发布，使用非常方便，同时自带一些语料处理工具，帮助用户训练自己的语料。

Hanlp 项目主页：

HanLP 下载地址：<https://github.com/hankcs/HanLP/releases>

Solr 5.x、Lucene 5.x 插件：<https://github.com/hankcs/hanlp-solr-plugin>

为了便于更好的学习使用 HanLP，你需要先获取到 Hanlp 的源码。这里我直接借助 IDEA 的 git 插件从 Github 仓库将 Hanlp 源码克隆到本地，如图 4-12 ~ 图 4-14 所示。

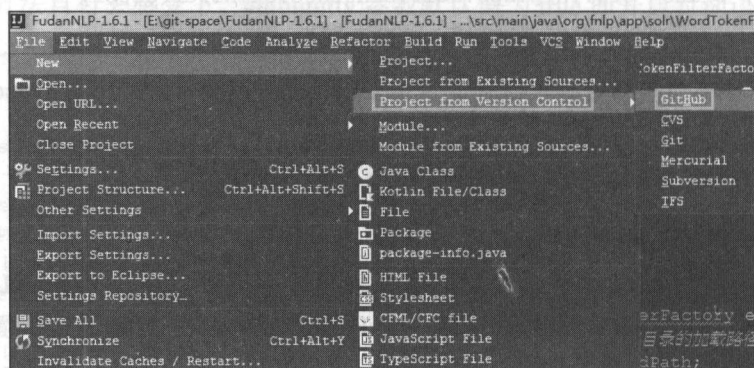


图 4-12 IDEA 中克隆 Github 上的项目源码到本地

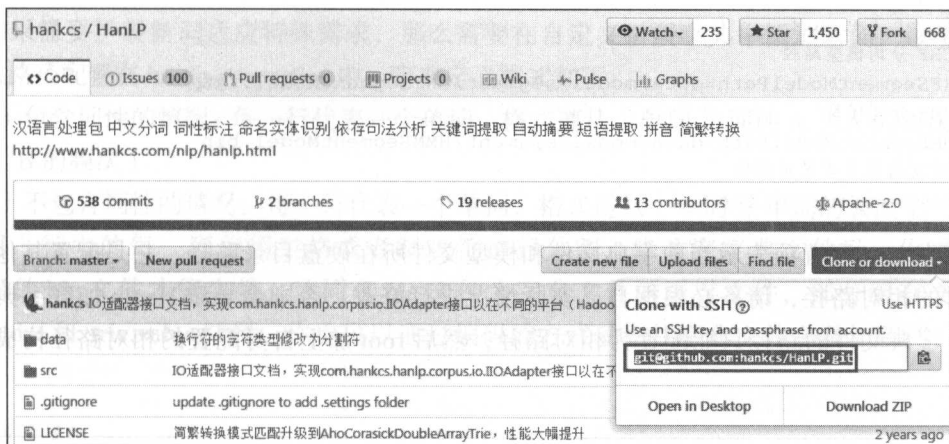


图 4-13 获取 Hanlp 项目的 Git 链接地址

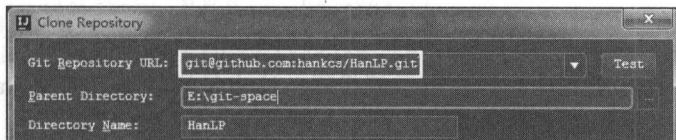


图 4-14 IDEA 中克隆 Git 项目

导入之后，首先你要查看项目的 classpath 路径下是否存在一个 hanlp.properties 属性配置文件，如果不存在，那么请手动创建并进行如下配置：

```
# 本配置文件中所有路径的相对根目录，根目录 + 其他路径 = 绝对路径
# Windows 用户请注意，路径分隔符统一使用 /
root=E:/git-space/HanLP
# 核心词典路径
CoreDictionaryPath=data/dictionary/CoreNatureDictionary.txt
# 2 元语法词典路径
BiGramDictionaryPath=data/dictionary/CoreNatureDictionary.ngram.txt
# 停用词典路径
CoreStopWordDictionaryPath=data/dictionary/stopwords.txt
# 同义词词典路径
CoreSynonymDictionaryPath=data/dictionary/synonym/CoreSynonym.txt
# 人名词典路径
PersonDictionaryPath=data/dictionary/person/nr.txt
# 人名词典转移矩阵路径
PersonDictionaryTrPath=data/dictionary/person/nr.tr.txt
# 繁简词典路径
TraditionalChineseDictionaryPath=data/dictionary/tc/TraditionalChinese.txt
# 自定义词典路径，用；隔开多个自定义词典，空格开头表示在同一个目录，使用“文件名词性”形式则表示这个词典的词性默认是该词性。优先级递减。
# 另外 data/dictionary/custom/CustomDictionary.txt 是个高质量的词库，请不要删除
CustomDictionaryPath=data/dictionary/custom/CustomDictionary.txt; 现代汉语补充词库.txt; 全国地名大全.txt ns; 人名词典.txt; 机构名词典.txt; 上海地名.txt ns; data/dictionary/
```



```

person/nrf.txt nrf
#CRF 分词模型路径
CRFSegmentModelPath=data/model/segment/CRFSegmentModel.txt
#HMM 分词模型
HMMSegmentModelPath=data/model/segment/HMMSegmentModel.bin
# 分词结果是否展示词性
ShowTermNature=true

```

其中 `root` 配置项表示当前字典文件和模型文件所在硬盘目录路径，它是配置其他字典文件时的相对路径，请务必根据自己实际情况进行修改，否则会导致 Hanlp 加载字典文件失败。字典文件的路径只能配置为相对路径，然后 `root` 加上自身配置的相对路径构成绝对路径。



**注意** `root` 参数可以配置为任意目录的绝对路径，不一定非得配置成当前项目的根目录。

然后你就可以直接运行 `src/test/java` 测试代码包下的示例程序进行学习并熟悉 Hanlp，其中几乎涵盖了 Hanlp 所有功能的测试示例代码，这也是用户学习使用 Hanlp 最好的资料，建议大家阅读并运行其中的每个 `demo`。

Hanlp 分词器类在 `com.hankcs.hanlp.tokenizer` 包下，默认提供了 8 种实现，分别用于不同的需求场景下。比如 `BasicTokenizer` 只进行基本的 NGram 分词，不识别人名、地名、机构名，不使用用户词典。`SpeedTokenizer` 极速分词，基于 Double Array Trie 实现的词典分词，适用于“高吞吐量”“精度一般”的场合。每个 `Tokenizer` 都提供了一个 `segment` 方法用于对指定的字符串进行分词，不过 Hanlp 对于所有功能都可以通过 Hanlp 工具类快速调用，这点还是蛮人性化的。

Hanlp 分词器核心部分是不依赖其他第三方类库的，Junit 除外，也就是说你可以很方便地将它用于任意 Java 项目中。在其他项目中使用它，你可以直接从 Hanlp 的 Github 上下载获取到 jar 包，如果你熟悉如何打 jar 包，你也可以自己编译打包。为了方便用户使用 Hanlp，作者提供了一个 Maven 配置，具体配置如下所示：

```

<dependency>
<groupId>com.hankcs</groupId>
<artifactId>hanlp</artifactId>
<version>portable-1.2.11</version>
</dependency>

```

但是这种方式有缺陷，CRF 分词和依存句法分析功能不可用，所以如果你有自定义需求，建议自己打 jar 并导入。借助 IDEA 或 Eclipse 打包其实很简单，但打包时需要注意的是，`data` 目录和 `src/test/java` 下测试代码请不要打到 jar 包内部。`data` 目录下存放的 Hanlp 需要加载的字典文件以及模型文件，不可随意删除，除非很清楚它的作用。你可以将 `hanlp.properties` 属性配置文件中的 `root` 配置项的参数值配置为 `data` 目录的硬盘路径。

如果需要扩展新词适应特殊需求,那么需要在自定义扩展字典中添加新词,该字典文件默认路径配置在 `hanlp.properties` 中,字典定义格式如下:

1) 包含词性的情况:每一行代表一个单词,格式遵从 [ 单词 ][ 词性 A ][ A 的频次 ][ 词性 B ][ B 的频次 ]。

2) 不包含词性的情况:每一行代表一个单词,格式遵从 [ 单词 ][ 单词的频次 ]。

每一行中的每一列使用空格或者制表符进行分割,修改完字典文件后,你需要删除同名的 `bin` 文件才能生效,该同名 `bin` 文件为字典文件的缓存文件,部分缓存文件是以 `.trie.dat` 和 `.trie.value` 结尾。你也可以通过编码的方式动态往字典文件中添加新词,示例如下:

```
// 动态增加
CustomDictionary.add(" 攻城狮 ");
// 强行插入
CustomDictionary.insert(" 白富美 ", "nz 1024");
// 删除词语 (注释掉试试)
CustomDictionary.remove(" 攻城狮 ");
CustomDictionary.add(" 单身狗 ", "nz 1024 n 1")
```

如果想要在 Lucene 和 Solr 中使用 Hanlp,你除了需要 `Hanlp.jar`,还需要下载 Hanlp 提供的 `solr-plugin`,其源码托管在 Git 仓库: <https://github.com/hankcs/hanlp-solr-plugin>。下载完成之后,请导入到 IDEA 或 Eclipse 中,查看源码进行熟悉,这也是熟悉如何在 Lucene 和 Solr 中使用 Hanlp 的最好方式,源码中也提供了很多使用示例。

在 Lucene 中,只需要创建 `HanLPIndexAnalyzer` 对象即可,你可以传入 `pstemming` 和 `filter` 参数,分别表示是否对英文进行词干还原(比如复数还原成单数、单词的时态还原)以及停用词。如果不传入 `pstemming` 参数,则表示不进行词干还原。同包下还提供了另一个 `HanLPAnalyzer` 实现,它跟 `HanLPIndexAnalyzer` 的区别是 `HanLPIndexAnalyzer` 是细粒度切分的,适合为对文本创建索引用于搜索,不过两者都是采用最短路径分词模式,最短路求解采用的是 Viterbi 算法,内部都是通过调用 `ViterbiSegment` 类来实现的。

在 Solr 中使用 Hanlp,你需要将 Hanlp 和 `han-solr-plugin` 两个项目分别打包成 jar 并复制到 `core/lib` 目录下,然后在 `core` 的 `schema.xml` 中配置 Hanlp 分词器的 `<fieldType>`,配置示例如下:

```
<fieldType name="text_cn" class="solr.TextField">
<analyzer type="index">
<tokenizer class="com.hankcs.lucene.HanLPTokenizerFactory"
enableIndexMode="true"/>
</analyzer>
<analyzer type="query">
<tokenizer class="com.hankcs.lucene.HanLPTokenizerFactory"
enableIndexMode="false"/>
</analyzer>
</fieldType>
```



HanLPTokenizerFactory 支持如表 4-5 所示的配置参数。

表 4-5 HanLPTokenizerFactory 配置参数

参数名	描述	默认值
enableIndexMode	启用索引模式	true
enableCustomDictionary	是否启用用户词典	true
customDictionaryPath	用户词典路径（绝对路径或相对 root 参数的相对路径，多个词典用空格隔开）	null
stopWordDictionaryPath	停用词词典路径	null
enableNumberQuantifierRecognize	是否启用数词和数量词识别	true
enableNameRecognize	开启人名识别	true
enableTranslatedNameRecognize	是否启用音译人名识别	false
enableJapaneseNameRecognize	是否启用日本人名识别	false
enableOrganizationRecognize	开启机构名识别	false
enablePlaceRecognize	开启地名识别	false
enableNormalization	是否执行字符正规化（繁体→简体，全角→半角，大写→小写）	false
enableTraditionalChineseMode	开启精准繁体中文分词	false
enableDebug	开启调试模式	false

4.3.9 Jieba 分词器

Jieba 分词器原版是使用 Python 开发的中文分词组件，这里说的是 Jieba 分词器的 Java 版，Java 版只保留的原项目针对搜索引擎分词的功能（cut\_for\_index、cut\_for\_search），词性标注（由于词性标注的性能有问题，最新版本中已经将这个功能去掉了），关键词提取没有实现。Jieba 分词器 Java 版支持两种分词模式：Index 模式和 Search 模式，Index 模式用于对文档进行索引，Search 模式用于对用户查询关键字进行分词。Jieba 分词器最大亮点是不需要额外的纠正词典就能实现消除歧义词，虽然它的 Java 版功能虽小，但短小精悍，如果你没有什么其他特别的分词需求，Jieba 分词器是可以考虑的。

Jieba 分词器采用的算法：

- ❑ 基于 Trie 树结构实现高效的词图扫描，生成句子中汉字所有可能成词情况所构成的有向无环图（DAG）；
  - ❑ 采用了动态规划查找最大概率路径，找出基于词频的最大切分组合；
  - ❑ 对于未登录词，采用了基于汉字成词能力的 HMM 模型，使用了 Viterbi 算法。
- Jieba 分词器源码托管在 Github，你可以执行 git clone 命令将其克隆到本地：

```
git clone https://github.com/huaban/jieba-analysis.git
```

你会发现 Jieba 分词器总共也就 8 个类，实现分词的核心类就是 JiebaSegmenter，再个就是字典类 WordDictionary，负责结巴分词的核心词典、用户自定义扩展字典等文件的加载与查询，毕竟结巴分词器也是基于字典匹配实现的即存在于字典文件中的词汇才会被分出

来，因此字典查询速度决定了分词性能。不过结巴分词器默认并不支持停用词过滤。由于结巴分词器的开发文档几乎没有，可能对初次使用结巴分词器的用户来说，会有点小困扰，比如 classpath 路径下的几个字典文件的用途。其中 dict.txt 和 dict.big.txt 属于结巴分词的核心词典，存在于该字典内的词将会被分出，dict.big.txt 词典支持繁体字，默认结巴分词器加载的是 dict.txt，即默认不支持对繁体词分词，如果你需要支持繁体字，那么你就需要修改源码使其默认加载 dict.big.txt，具体到代码层面就是你需要修改 WordDictionary 类的 MAIN\_DICT 常量值为 dict.big.txt。再个就是 prob\_emit.txt 模型文件，它是用于实现新词发现功能的，涉及的类为 FinalSeg，结巴分词器的新词发现功能是基于 HMM 模型使用 Viterbi 算法实现，prob\_emit.txt 即依赖的模型文件，FinalSeg 类在构建时需要加载此文件，默认是从 classpath 路径查找该文件。然后就是项目根目录下的 conf 目录中提供了有两个字典文件：user.dict、sougou.dict。它们是用户自定义扩展字典，用于对核心词典的扩充。将分词器不能分出来的词语添加到扩展字典中，即可以实现对新词进行分词。

使用结巴分词也非常简单，你只需要调用 JiebaSegmenter 类的 process 方法，传入需要分词的字符串以及 SegMode 分词模式，分词模式默认有 INDEX 和 SEARCH。INDEX 模式用于对文本进行创建索引时使用，它属于细粒度切分，SEARCH 模式用于对用户搜索关键词进行分词，属于粗粒度的空格字符或标点符号短句方式切分。不过遗憾的是结巴分词器 Java 版并没有集成 Lucene 和 Solr，即你不能在 Lucene 和 Solr 中使用结巴分词器，为此我进行了扩展，使其支持用户自定义扩展字典、停用词字典功能，并且字典文件和模型文件的加载支持多种路径，不再是默认的绝对路径或 classpath 路径下，而是尝试在 classpath、jar 包内部、项目根目录、以及用户指定的硬盘路径下进行多次加载尝试，修改后的文件加载机制有 FileUtils 工具类提供。此外，字典文件和模型文件的加载路径支持可配置化，你只需要在项目的 classpath 路径下添加一个 Jieba.xml，并按照里面的注释进行配置，使用结巴分词器显得更便捷了。同时，我将其集成了 Lucene&Solr5.x，方便大家在 Lucene 和 Solr 下使用结巴分词器。改造后的结巴分词器在我的 Github 仓库，代码下载地址如下所示：

<https://github.com/yida-lxw/jieba-analysis>

在 Lucene 中使用结巴分词器，你需要创建一个 JiebaAnalyzer 分词器对象，你可以传入一个表示分词模式的 mode 参数，该参数也可以借助一个 SegMode 枚举方式传入，该枚举可选值有：

```
public enum SegMode {
    DEFAULT {
        public String getValue() {
            return MODEL_DEFAULT;
        }
    },
    INDEX {
        public String getValue() {
```

```

        return MODEL_INDEX;
    }
},
SEARCH {
    public String getValue() {
        return MODEL_SEARCH;
    }
},
QUERY {
    public String getValue() {
        return MODEL_QUERY;
    }
};
public abstract String getValue();
public static final String MODEL_INDEX = "index";
public static final String MODEL_DEFAULT = "default";
public static final String MODEL_SEARCH = "search";
public static final String MODEL_QUERY = "query";
}

```

分词模式的枚举默认只有 INDEX 和 SEARCH，我将其扩展到了 4 个，其实 DEFAULT 和 INDEX 是一对，SEARCH 和 QUERY 是一对。如果你想要配置自定义扩展字典和停用词字典，你只需要在 Jieba.xml 中进行配置并将字典文件复制到你配置的目录下即可。

如果你想要在 Solr 中使用结巴分词器，只需要将结巴分词器的源码打包成 jar 并将其复制到当前 core/lib 目录下，然后将 Jieba.xml 复制到 \${TOMCAT\_HOME}/webapps/solr/WEB-INF/classes 目录下，并编辑 Jieba.xml 配置你的自定义扩展字典、停用词字典以及模型文件的加载路径，如果你不需要可以不配置，同时将文件放置到配置的路径下。最后需要在当前 core 的 schema.xml 中配置你的 <fieldType>，下面是一个简单的配置示例：

```

<fieldtype name="text_jieba" class="solr.TextField">
  <analyzer>
    <tokenizer
      class="com.huaban.analysis.jieba.solr.SentenceTokenizerFactory" />
    <filter class="com.huaban.analysis.jieba.solr.JiebaTokenFilterFactory"
      mode="index" />
  </analyzer>
</fieldtype>

```

#### 4.3.10 分词器使用建议

分词器核心就是算法和字典，算法决定了单位时间内的分词效率，字典对于基于字典匹配实现分词的分词器来说至关重要，词库不完善可能就导致词语分不出来，虽然你可以通过自定义扩展词典去补充，但补充只能是发现问题后再去补充，所以分词器自动发现新词的功能就显得尤为重要。IK 分词器虽然提供了一个 useSmart 参数表示是否开启智能分词，然而这根本谈不上“智能”二字，仅仅是依据字典匹配。CRF 方法是目前公认的效果最好的

识别新词的分词算法，不过它需要你根据语料库不断训练。当你需要精准分词时，可以考虑 CRF 分词，比如 Ansj 和 HanLP 都支持 CRF 分词。Ansj 的亮点是加载核心字典时采用双数组 Trie 树算法，但 Ansj 的硬伤是核心字典不能修改，只能借助自定义扩展字典来补充。对于 HanLP，我几乎看不到它什么软肋，功能比较齐全，非常强悍，而且社区也很活跃，从 Github 上该项目被 Star 喜爱的人数就知道了。其实最让我惊叹的是结巴分词器，区区 8 个类实现的效果着实惊艳到我了，短小精悍，麻雀虽小但五脏俱全啊。基于 HMM 模型使用 Viterbi 算法实现的自动新词发现功能（即智能分词）是它的亮点，而且在消除歧义词方面，结巴分词器也表现不俗，Ansj 虽然也支持消除歧义词，但 Ansj 需要借助一个额外的纠正字典文件，然而结巴分词器不需要，确实厉害。比较遗憾的是，结巴分词器不支持词性标注，在功能方面就仅仅只有分词了，没有 HanLP 和 Jcseg 分词器支持的功能多。如果没有特殊的需求，那么你可以考虑下结巴分词器。不过在中文分词领域，中科院的 Ictclas 绝对是最权威的，在 Java 中使用它，你只能通过 JNA 调用 DLL 来实现，有点隔靴搔痒的感觉，非常不爽，最制约你的估计是它是收费的并不开源，你只能拿它当玩具，有兴趣的同学可以学习研究一番，但实际项目中应用需要付费。由于不是基于 Java 开发的，所以对于 Java 程序员来说，项目中选择 Ictclas 商业版并做二次开发，风险会比较大，不建议使用。复旦大学的分词器 FudanNLP 跟 HanLP 相比，几乎无亮点，甚至 HanLP 在某些方面更出色。再个比较相似的就是 MMseg4J 和 Jcseg，两者使用了 MMseg 算法，但明显 Jcseg 更惹人注目，而且 MMSeg4J 貌似已经很久没更新了。IK 分词器应该是使用者比较多的一种分词器了，我想很大一部分原因是它的算法足够简单，很容易进行扩展改造，字典文件支持编程方式自由加载，虽然不支持字典文件更新自动检测，但你可以自己修改源码对其进行改造，IK 最大的弊端是对消除歧义词处理的不够理想，这也是庖丁解牛分词器的最大软肋，虽然庖丁解牛字典查找效率极高。Paoding 最大亮点是字典和配置文件分类细致，可扩展性和可维护性极强。IK、Jieba、Paoding 这 3 款分词器虽然各有缺点，但它们是相对比较容易修改源码进行的扩展的分词器，可控性比较强，即便使用过程中遇到了什么坑，可以自己调试源码然后自己填坑，使用风险低，如果你对分词效果不是很敏感，追求简单适用，那么你可以考虑 IK、Jieba 或者 Paoding。

分词器的分词效率我觉得没有比较的意义，分词再快分出来的效果不满足搜索需求也是白搭，所以一切要以实际需求来决定。分词效率只决定你创建索引需要耗时多久，索引的最终目的是为了搜索，搜索功能是最终面对用户的，想要用户使用体验好，那么就要求你分词后建立的索引体积要尽量小，而且分词效果能满足用户需求，比如“团购网站”，用户的输入的搜索关键字可能是“团购”、“团购网”、“网站”、“团购网站”，那么如果你分词分出来的效果是“团购”、“网站”、“团购网站”，那么用户输入“团购网”关键字就搜不到内容。那些支持 CRF 分词的分词器在这方面会有优势，但需要根据语料库进行大量的训练，这需要投入人力成本，而 IK 虽然效果不佳，但你可以通过自定义扩展字典实现或者你通过 NGram 方式解决，显然使用 IK 更省时省力，这也是大部分公司考虑的一方面。如果你追求

简单，使用风险低，没有其他额外需求比如需要支持人名地名自动识别，新词自动识别而不是人工维护扩展词典、支持歧义词消除等，那么你可以考虑使用 IK 或 Jieba，如果你想要两者兼顾，那么可以选择 HanLP，而且 HanLP 社区很活跃，如果你对分词效果要求很高又不差钱，那么可以选择购买 Ictclas 的商业授权。

## 4.4 本章总结

在本章中，我们首先学习了分词的基本概念，比如如何理解 Analyzer、Tokenizer、TokenFilter。理解这些概念是你能够使用 Solr 分词器的前提，否则在使用 Solr 分词器过程中一遇到问题就会手足无措。如果你想要能够自己修改分词器的源码以满足自己独特的需求，那么更有必要熟悉分词的基本概念以及分词过程中的整个调用链。然后我们需要学习 Solr 内置提供的各种分词器，由于官方提供的分词器对中文并不友好，因此最后我们着重详细了解了目前市面上常见的各种中文分词器，大家可以按照自己的具体喜好有选择性的阅读。

## Solr 查询

通过第 5 章，你将可以学到如下内容：

- Solr 查询相关理论；
- Solr Relevance 相关理论；
- Solr 内置的各种查询解析器（如 DisMax、eDisMax）；
- Solr 中 Query 与 Filter Query 的区别；
- Solr 如何返回查询结果集；
- Solr 如何分页查询；
- Solr 排序；
- Solr 查询的调试模式。

### 5.1 Solr 查询概述

Solr 提供了一系列丰富且灵活的查询功能，为了更好地理解 Solr 查询的灵活性，先从宏观上了解 Solr 的查询组件会是一个很好的开端。

当用户发起了一个 Solr 查询请求，这个查询请求会被 Request Hanlder 接收并处理。Request Hanlder 是 Solr 的内置插件，用于定义 Solr 处理用户请求的逻辑。Solr 支持多种 Request Handler，一些是为了处理查询请求而设计，而另一些 Request Hanlder 则用于管理一些 Solr 任务，比如索引复制。

默认情况下，Solr 会选择一个特定的 Request Handler 来处理用户的查询请求，除此之外，Solr 还允许用户通过配置的方式来选择一个自己喜好的 Request Handler 来覆盖 Solr 的



默认选择。用于处理 Solr 查询请求的 Request Handler 被称为 Query Parser，它是用于解析用户传入的查询关键词以及查询参数的。不同的 Query Parser 支持不同的查询语法，Solr 默认的 Query Parser 是 StandardQueryParser（即标准查询解析器），它其实是继承自 Lucene 的 Query Parser。Solr 同时还包含 DisMax Query Parser 和扩展的 DisMax Query Parser。标准查询解析器对于用户传入的查询表达式要求比较严格，如果用户输入的查询表达式不符合规范，可能会抛出异常，而不是返回空结果集。而 DisMax Query Parser 则有更高的查询表达式错误容忍度，即便查询表达式语法错误也不会抛出异常。DisMax Query Parser 设计的目的是为了提供了一个类似当前那流行的搜索引擎，比如 Google 的使用体验，而扩展的 DisMax Query Parser（即 eDisMax Query Parser）则是 DisMax Query Parser 的功能改进版，它除了兼容 Lucene 的查询表达式语法以及错误容忍，还包含几个额外的特性。

Query Parser 可以接收用户以下输入：

- 查询搜索关键词：即存在于索引中用于查询的 Term；
- 为了调整查询，在查询的 Term 之间通过使用 boolean 运算符，增加某些特定字符串，增加域的重要性或从查询结果中排除一些内容而传递的参数；
- 指定控制查询响应表示形式的参数，比如 wt 参数；用于指定返回结果集顺序的排序参数，比如 order 参数；用于限制响应中返回指定的部分 Field 时传递的参数，比如 fl 参数等。

你还可以指定一个 filter query 参数，作为查询响应的一部分，Filter Query 会根据整个索引数据和缓存的结果集发起一个查询，因为 Solr 会为每个 Filter Query 都分配一个独立的缓存，这种策略可以提高 Solr 的查询性能。（注意：尽管 Filter Query 与 Analysis Filter 有着相似的名称，但它们并不相关）Filter Query 在查询时会根据已经存在于索引中的所有索引数据执行一个 Solr 查询，而 Analysis Filter（比如 Tokenizer）会遵循指定的规则解析文本内容用于索引创建。

一个 Solr 查询请求的返回响应中的部分 Term 可以被高亮突出显示，那些被选中的 Term 会在带有颜色的 CSS 盒子模型中显示，以便于用户能一下子被屏幕上的搜索结果所吸引。然而 Highlighting 功能使得实现在返回的长文本找出相关信息变得非常简单。Solr 的查询响应支持配置高亮摘要片段，流行的搜索引擎比如 Google 会在他们的查询结果中返回摘要片段，所谓摘要片段就是使用 3 ~ 4 行的文本描述当前返回的查询结果。

为方便首次进入搜索界面的用户找到他们想要的内容，Solr 支持两个特殊方式实现分组查询，即 Faceting（分类）和 clustering（聚合），从而引导用户接下来的页面跳转。

Faceting 是根据索引的 Term 对查询结果集分配到不同分类下的一个过程，在每个分类中，Solr 会统计每个相关 Term 下的命中结果数量，这个命中结果数量在 Facet 中被称作 facet constraint。

Faceting 使用户可以非常方便地浏览查询结果，比如电影网站、商品网站，他们会有很多分类，每个分类下可能有很多子分类和商品。网站可以统计每个分类下商品的数量，依据



提供的分类下商品的数量大小，能够一定程度上引导用户的下一步单击行为，比如大部分用户会选择单击分类下商品数目多的分类进行下一步商品搜索。Faceting 需要使用搜索程序创建的索引中的 Field，这些 Field 可以用于对索引中每条数据进行分类，那么它就可以用于 Faceting，比如手机商品中可能会有 brand、屏幕尺寸、网络制式、颜色，甚至手机的价格区间这些数据信息，它们都可以作为 Faceting 分类统计的域。Faceting 设计的目的就是实现用户导航，并提供一定的分组统计，最终改善用户体验。

Clustering 会根据结果集的相似度在执行查询时对结果集进行自动分组，而不是在索引创建时分组。Clustering 返回的结果集通常缺乏一种优雅的层级结构来用于在 Faceted 查询结果集中查找。尽管如此，Clustering 还是很有用的，它可以为用户找出查询结果集之间意想不到的共性，可以帮助用户排除一些他们不关心的查询结果集。

Solr 还支持 MoreLikeThis 功能，它支持根据给定的查询关键字找出与结果集相似的其他结果集。它能够为用户提供更多用户可能会关心的相似信息，引导用户继续单击发起一个新的查询请求。

Solr 还提供了 Response Writer 组件用于管理查询响应数据的输出格式，Solr 提供了多种 Response Writer，包括 XML Response Writer 和 JSON Response Writer。

图 5-1 总结了 Solr 查询过程中的一些关键点。

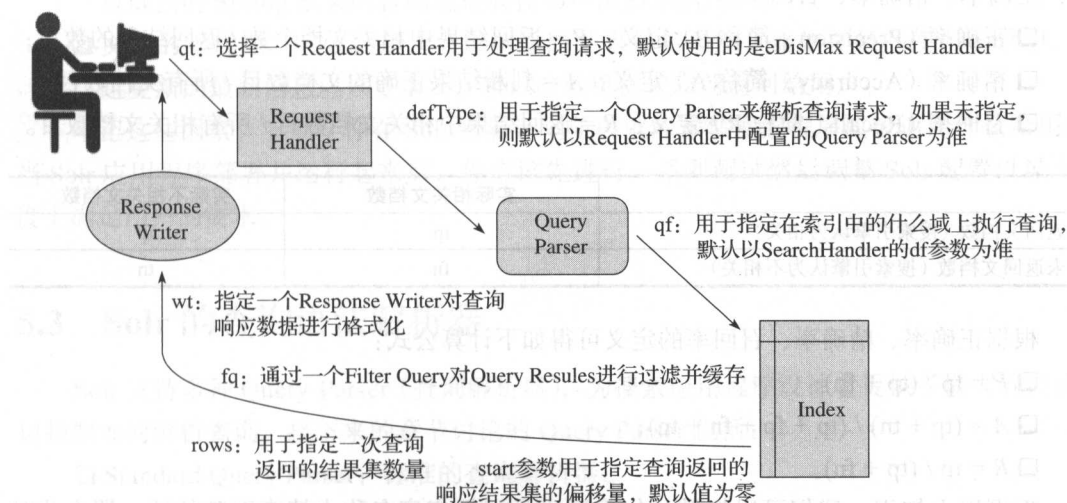


图 5-1 Solr 查询处理过程示意图

## 5.2 Solr 查询相关度简述

Relevance 是用于表示查询响应满足用户查询需求的一个程度。查询响应的相关性程度取决于该查询的执行场景。一个搜索程序不同的使用场景下用户可能会有不同的需求和期

望，比如对于一个爱好电影的用户，那么应该尽量给用户返回更多跟电影相关的信息，而对于一个搞 IT 的码农，应该尽量返回跟编程相关的技术文章或书籍。即便是相同的查询关键字，搜索程序也应该根据不同用户作出不同的响应。比如，搜索“Spring”肯定本意是想查找跟 Spring Framework 相关的信息，而对于爱好音乐的用户，可能就是想要查找歌名中包含“Spring”的歌曲。

因为用户的搜索动机是变化的，所以同理，同一个查询响应的相关性也是变化的。那么应该怎么去综合衡量或评价一个查询响应呢？一般来说，与 Relevance 类似，这个问题的答案取决于查询的使用场景，在某些场景下，一个搜索查询在响应结果中能过滤掉一些不相关的索引文档是非常有价值的，比如一个用户根据疾病名称希望搜索能治疗该疾病的好医院，那么返回的结果中不包含不相关的医院显得很有意义。而对于一个电商网站，用户想搜索一双鞋子，如果返回的结果中包含了不相关的衣服信息也并无大碍。当我们配置 Solr 服务时，除了应该考虑 Solr 搜索响应结果集的相关性之外，还应该根据其他因素来全面衡量你的搜索效果，比如搜索的实时性、易用性、索引构建时的系统开销等。从用户的角度去评价一个搜索引擎的搜索相关度最好的方法就是计算用户搜索到自己想要的信息时已经浏览了多少文档。但是实际情况是，用户的查询动机是千变万化的，索引数据也是不断更新变化的，所以这种衡量方式不可取。于是，人们便提出了下面 3 个概念，并建立了一个评价标准：正确率、精确率、召回率。

- 正确率 (Precision, 简称 P) 定义： $P = \text{返回结果中相关文档个数} / \text{返回结果的数目}$ ；
- 精确率 (Accuracy, 简称 A) 定义： $A = \text{判断结果正确的文档数目} / \text{所有文档数目}$ ；
- 召回率 (Recall, 简称 R) 定义： $R = \text{返回结果中相关文档数目} / \text{所有相关文档数目}$ 。

	实际相关文档数	实际不相关文档数
返回文档数 (搜索引擎认为相关)	tp	fp
未返回文档数 (搜索引擎认为不相关)	fn	tn

根据正确率、精确率、召回率的定义可得如下计算公式：

- $P = tp / (tp + fp)$ ；
- $A = (tp + tn) / (tp + fp + fn + tn)$ ；
- $R = tp / (tp + fn)$ 。

根据以上知识，我们可以知道，如果用户是根据疾病名称去搜索医院的话，那么此时的召回率（即所有相关的文档中搜索引擎返回了多少相关文档给用户）应该尽量高，因为假如返回的是虚假骗的医院，对用户来说就是坑财害命的。但此时对于正确率要求并不高，正确率高在这个使用场景下，也没有多大的意义，因为返回的结果中相关性高的医院有 100 家，与返回有相关性的医院 10 家没太大差别，用户实际也只可能衡量比较前几家医院最后选择其中一家，返回太多反而影响用户的选择。因此在某些使用场景下，返回具有高相关性的少量查询结果或许是最好的方式。

利用正确率、精确率、召回率这几个概念，可以量化用户查询返回的结果的相关性，一个完美的搜索系统应该对于每个用户的每个查询拥有 100% 的正确率和 100% 的召回率，换句话说，它会返回所有具有相关性的文档给用户。一般来讲，在真实的系统中，当我们谈论正确率和召回率时，我们通常只关注一定数量的结果集上的正确率和召回率，最常见的就是我们一般只返回一页的结果集，比如 10 条、20 条，而不是一次性返回全部的查询结果集。

为了返回更有相关性的结果给用户，Solr 可以通过 Faceting、Filter Query 以及其他查询组件进行灵活的配置，在正确率和召回率之间获取平衡，以帮助用户调整它们的查询，从而满足特定用户群体的需求。因此配置 Solr 应该考虑以下几个方面：

- ❑ 应用程序的各种用户的需求，包括易用性和响应速度，一些特殊比较严格的需求除外；
- ❑ 对于某些用户来说，对数据进行有效分类会非常有意义，比如用户购买手机，比较关心的是手机的品牌、价格、系统内核、像素等，那么根据这些信息进行分类，可以方便以后找到他们想要的东西；
- ❑ 应该将一些比较权威性的数据信息尽量提到前面显示，比如一个程序员搜索“Spring”，那么显而易见他是希望查找 Spring 框架相关信息，此时应该优先将更有权威性的 Spring 框架的官网地址放在第一位展示给用户；
- ❑ 文档的年代性、时效性是否会影响返回结果集相关性，比如在某些使用场景中，最近更新的最新信息应该更重要，所以应该更优先靠前返回给用户。

牢记这些因素，有助于你在 Solr 部署计划阶段就能预估搜索程序应该返回什么给用户，当 Solr 应用程序部署并运行起来后，你应该先进行一系列测试然后调整 Solr 配置以最大程度上满足用户的需求。

### 5.3 Solr 的查询语法解析器

Solr 支持多种 Query Parser（查询解析器），为搜索应用程序设计者提供了极大的灵活性以控制如何解析查询。接下来的章节讨论的 Query Parser 包括：

- ❑ Standard Query Parser：标准的查询解析器；
- ❑ DisMax Query Parser：DisMax 查询解析器；
- ❑ Extended DisMax Query Parser：扩展的 DisMax 查询解析器；
- ❑ Other Parsers：其他 Parser。

Solr 的 Query Parser 插件都继承自 QParserPlugin 类，如果你有自定义 Query Parser 的需求，那么需要继承 QParserPlugin 类以创建你自己的 Query Parser 插件。

Query Parser 是一个负责解析查询文本并将其转换成 Lucene 里的 Query 对象的组件。有多种可选的方式来为一个查询请求指定使用哪个 Query Parser。

defType 参数用于指定查询默认使用什么 Query Parser，使用示例：

```
&q=foo bar&defType=lucene
```

使用 LocalParams 语法来指定使用什么 Query Parser，使用示例：

```
&q={!dismax}foo bar
```

所谓 LocalParams 即代表了一些本地参数，提供了一种将元数据信息添加给特定的参数类型（比如查询文本）的方式，LocalParams 总是以前缀参数的表现形式传递给 Solr，比如你有这样的一个查询：

```
q=solr rocks
```

我们可以在查询文本的前面使用 LocalParams 添加前缀参数来提供更多的信息给 Query Parser，比如改变默认的 Boolean 操作符为 AND 以及改变默认的查询域为 title：

```
q={!q.op=AND df=title}solr rocks
```

指定 LocalParams 的语法是，在查询文本的前面添加一对花括号作为前缀，然后花括号内部以感叹号“!”开头，然后后面是使用空格分割的多个 key = value 的键值对，因此假如原始的查询文本是 foo，那么应用了 LocalParams 之后的查询文本可能是这样的：

```
{!k1=v1 k2=v2 k3=v3}foo
```

注意：每个查询文本只能指定一个 LocalParams，花括号内的 key-value 键值对中的 value 可以使用单引号或双引号进行包裹，比如查询多个域需要使用 qf 参数，你可以这样指定：

```
q={!type=dismax qf='myfield yourfield'}solr rocks
```

如果一个 LocalParams 的 key-value 键值对缺少 key 只有 value，那么默认隐式的 key 为 type，这允许用户以简短的形式来指定 type 参数，所以：

```
q={!dismax qf=myfield}solr rocks
```

等价于

```
q={!type=dismax qf=myfield}solr rocks
```

LocalParams 内部的特定 key：“v”，提供了一种可选的方式用来指定查询的文本即用户输入的查询关键字，因此：

```
q={!dismax qf=myfield}solr rocks
```

等价于

```
q={!type=dismax qf=myfield v='solr rocks'}
```

LocalParams 内部的 key-value 键值对中的 value 值可以引用自一个外部变量，而不需要直接为 value 指定值，这个特性可以用来简化查询，可以用来分离用户输入的查询参数或者分离那些来自 solrconfig.xml 中默认配置的前端界面传递的参数。因此：

```
q={!dismax qf=myfield}solr rocks
```

等价于

```
q={!type=dismax qf=myfield v=$qq}&qq=solr rocks
```

上面的 \$qq 表示一个变量，它定义在 LocalParams 的外部，变量名必须以 “\$” 开头，变量名称遵循命名规范即可，也就是说，你可以任意定义多个外部变量将用户输入的查询文本进行分离。

Solr 提供了很多内置的开箱即用的 Query Parser，它们是：

- lucene：即默认的 Lucene 里的 Query Parser。
- dismax：DisMax parser 允许查询跨多个域并使用不同的权重。
- edismax：ExtendedDisMax parser 是 edismax 的升级版，提供了更多特性。
- maxscore：MaxScore Query Parser 表示 Boolean 查询中通过 Boolean 操作符连接的多个子查询的最大评分作为最终的得分，默认是使用 sum 求和。（自 Solr 4.4 版本开始可用）。
- func：即 FunctionQParser，通过 Solr 内置的或自定义的功能函数来干预文档评分。
- boost：通过 Function Query 为一个 Query 查询添加权重。
- frange：即 FunctionRangeQParser，它主要用于对 Function Query 进行范围过滤，示例如下：  
fq = {!frange l = 0 u = 2.2}sum (user\_ranking, editor\_ranking)。
- field：用于构造 FieldQuery 的 Query Parser，其实 FieldQuery 都可以使用类似这样的查询表达式进行表示：title: foo，而你也可以这样指定：{!field f = title}foo，两者是等价的，这里的 f 是固定的 key，表示域的名称。
- prefix：用于构造 PrefixQuery 前缀查询的 Query Parser。使用语法如下所示：  
{!prefix f = myfield}foo，跟 Lucene 里的 myfield: foo\* 写法是等价的，都是表示在 myfield 域上查询以 foo 开头的索引文档。
- raw：用于构造 TermQuery 的 Query Parser。使用语法如下所示：  
{!raw f = myfield}Foo Bar，跟 Lucene 里的 TermQuery (Term("myfield", "Foo Bar")) 写法是等价的。Solr 4.0 版本开始，使用 term 来代替 raw。
- term：用于构造 TermQuery 的 Query Parser。用法与 raw 一样。
- surround：即 SurroundQParser，用于构造 SpanQuery 的 Query Parser，使用示例如下所示：

`fq = {!surround}fieldName: 2w (foo, bar)`，这里 `w` 表示经过移动几次能得到 “foo, bar”，同理还有个 `n`，`n` 是没有顺序的，即移动后得到 “bar, foo” 也是满足的，而 `w` 是有顺序的。`2w` 即表示移动 2 次后得到 foo, bar 且顺序也要保证。

- ❑ `simple`：即 `SimpleQueryParser`，自 Solr 4.7 版本开始可用。
- ❑ `complexphrase`：即 `ComplexPhraseQueryParser`，自 Solr 4.8 版本开始可用。
- ❑ `query`：表示 `Nested Query Parser`，用于构造 `Nested Query`（嵌套查询），它允许用户通过一些参数设置来重新定义一个查询的类型，比如将一个 `Query` 转换成其他类型的 `Query`，示例如下：

`_query_: {!query defType = func v = $q1}` 表示将一个 `Query` 转换成 `Function Query`，作用于 `$q1` 变量表示的域上。如果是这样表示：`_query_: {!lucene}inStock: true`，那么就跟默认的 `Lucene Query Parser` 是等价的。

虽然 Solr 内置了这么多种 `Query Parser`，但有些 `Query Parser` 拥有一套通用的查询参数。表 5-1 总结了 Solr 中的通用查询参数，`Standrad`、`DisMax`、`eDisMax Query Parser` 都支持。

表 5-1 Solr 中的通用查询参数

参 数	描 述
<code>defType</code>	指定 <code>Query Parser</code> 的类型
<code>sort</code>	指定响应结果集的排序规则
<code>start</code>	指定返回的响应结果集的偏移量，默认值为零
<code>rows</code>	指定一次查询返回多少个索引文档，一般用于分页
<code>fq</code>	即 <code>filter query</code> 的缩写，表示对查询结果集再发起一次过滤查询
<code>fl</code>	指定返回的响应结果集中包含哪些域，注意：指定返回的域必须是 <code>stored = true</code> 或者 <code>docValues = true</code>
<code>debug</code>	用于指定是否在响应结果中返回调试信息，若 <code>debug = query</code> 则会返回 <code>Query Parser</code> 相关信息，若 <code>debug = timing</code> 则会返回查询的各个阶段的耗时情况，若 <code>debug = results</code> 则会返回查询的执行计划相关信息
<code>explainOther</code>	用于在一个主查询中执行一个查询计划，比如： <code>q = bookName:java&amp;debugQuery = true&amp;explainOther = id:1</code>
<code>timeAllowed</code>	指定查询响应的最大超时时间
<code>segmentTerminateEarly</code>	表示是否在段文件合并之前对文档进行排序
<code>omitHeader</code>	表示在响应结果中不返回 <code>responseHeader</code> 信息
<code>wt</code>	即 <code>write type</code> 的缩写，用于指定使用什么类型的 <code>Response Writer</code> 来格式化查询响应结果
<code>logParamsList</code>	用于指定对哪些请求开启日志，多个参数使用逗号分隔
<code>echoParams</code>	用于指定是否打印请求参数信息

(1) `defType`

此参数用于指定使用什么类型的 `Query Parser` 来解析 `q` 参数，从而生成一个 `Query`。配置示例如下：



defType=dismax

如果 defType 参数未指定，那么默认会使用 Standard Query Parser 即 defType = lucene。

(2) sort

sort 参数用于对查询结果集按照升序或降序的方式进行排序，你可以使用此参数对数字或文本字符串进行排序，排序方式可以使用 ASC 或 DESC 来指定（不区分大小写）。Solr 支持对查询返回的响应结果集按照文档的评分进行排序或者域的域值进行排序，前提是该域 indexed = true 或者 docValues = true 并且 multiValued = false。

表 5-2 解释了 Solr 是如何处理各种 sort 参数设置的。

表 5-2 Solr 对 sort 参数的设置

示 例	解 释
—	如果 sort 参数未显式指定，那么默认是按照文档的评分降序排列即 score desc
score desc	显式的指定按照文档的评分从高到低进行排列
price asc	显式指定按照 price 域的域值进行升序排列
inStock desc, price asc	显式指定先按照 inStock 域降序排列，再按照 price 域升序排列

在指定排序方式时需要注意以下两点：

- 在指定排序方式时，需要在 ASC 或 DESC 之前添加域名，或者使用 score 这个伪域也可以；
- sort 参数的指定格式为 sort = <field name> + <direction>, <field name> + <direction>], 多个排序规则之间使用逗号分隔或者使用 20% 分隔也行。

(3) start

当为一个查询结果集指定了 start 参数即表示告诉 Solr 从 start 参数指定的偏移量位置开始显示结果集。start 参数若未显式指定，那么默认值为零。当 start 参数指定为其他数字，比如 3，那么 Solr 会跳过结果集中的前 3 条。

你可以使用 start 参数用于分页，比如假设 rows 参数设置为 10，start 参数设置为 5，那么会返回 5 ~ 14 之间的索引文档。记住，start 参数是从零开始计算的。

(4) rows

你可以使用 rows 参数对 query 查询进行分页，此参数用于指定 Solr 一次返回的最大文档数量。rows 参数若未显式指定，那么默认值是 10。

(5) fq

fq 参数用于定义一个 Filter Query、Filter Query 用于对普通的 Query 查询返回的结果集进行一次过滤，由于 Filter Query 并不进行文档评分操作，所以它对于提升复杂查询的性能非常有用。Filter Query 拥有独立主查询的缓存，当后续的一个查询使用了当前这个相同的 Filter Query，那么会直接命中缓存，被过滤的结果集会快速从缓存中返回。

当你使用 fq 参数时，你需要始终记住以下几点：



□ fq 参数可以在一个 query 被指定多次，Filter Query 与 Query 之间的交集部分的索引文档会被过滤掉，下面是一个简单的使用示例：

```
fq=popularity:[10 TO *]&fq=section:0
```

□ Filter Query 也可以涉及复杂的 Boolean 查询，上面的示例也可以改写成一个单一的 fq，比如下面这样：

```
fq=+popularity:[10 TO *] +section:0
```

□ 每个 Filter Query 都有一个独立的缓存区，所以，如果这两个条件是经常一起出现的，你应该使用同一个 fq，如果两个条件是相对独立的，那么你应该使用两个 fq。但是需要注意的是，如果你将多个条件使用一个 fq 表示，那么意味着对于这多个条件过滤返回的结果集全部存储在一个缓存区中，有可能你的缓存区大小不够，会导致缓存无法命中，降低了 Filter Query 的查询性能。虽然你可以通过在 solrconfig.xml 中配置 filterCache 调整缓存区的大小，但应该思考的是你的 Filter Query 缓存区应该尽量充分利用，分配了 500M 实际每次只利用了 50M，那就是浪费；

□ 在 Solr 查询请求 URL 中指定的所有参数涉及的特殊字符都应该进行转义或编码成十六进制。

(6) fl

fl 参数通过指定一个域列表来限制查询响应的结果集中应该包含的域，指定的域应该满足 stored = true 或者 docValues = true。fl 参数可以指定多个域，多个域名称之间通过逗号或者空格分割。你可以在 fl 参数中将“score”作为一个隐含的域添加进去，这样响应结果集中就会包含每个文档的具体得分。如果你指定 fl 参数等于一个通配符“\*”星号，那么表示返回 Document 中的所有 stored = true 或 docValues = true 的域，并且 useDocValuesAsStored = true（当启用 docValues 时 useDocValuesAsStored 自动启用）。你还可以添加一些“伪域”到 fl 参数中，比如 function、transformer。

如表 5-3 演示了如何指定 fl 参数的一些基本示例。

表 5-3 指定 fl 参数示例

示 例	解 释
id name price	只返回 id, name, price 这 3 个域
id, name, price	只返回 id, name, price 这 3 个域
id name, price	只返回 id, name, price 这 3 个域
id score	只返回文档的 id 域以及文档的评分
*	返回文档中所有 stored = true 或 docValues = true 且 useDocValuesAsStored = true 的域，这也是 fl 参数的默认值
* score	返回文档中所有 stored = true 或 docValues = true 且 useDocValuesAsStored = true 的域，同时返回每个文档的评分
*,dv_field_name	返回文档中所有 stored = true 或 docValues = true 且 useDocValuesAsStored = true 的域，同时返回 dv_field_name 域的 docValues，尽管它的 useDocValuesAsStored = false

响应结果集中的每个索引文档可以应用一个 Function 函数，函数的计算结果可以作为一个“伪域”添加到 fl 参数中，示例如下：

```
fl=id,title,product(price,popularity)
```

Document Transformers 可以用于修改返回的响应结果集中的每个索引文档的信息，你可以将 Document Transformers 处理的信息作为一个伪域添加到 fl 参数中，示例如下：

```
fl=id,title,[explain]
```

你还可以为 fl 参数中的每个域，“伪域”或 Function、Document Transformer 定义一个别名，就像关系型数据库的 SQL 语句里可以为 select 查询语句中返回的字段起个别名，这样响应结果里显示的就不是具体的域名而是定义的别名啦。示例如下：

```
fl=id,sales_price:price,secret_sauce:prod(price,popularity),why_score:[explain style=nl]
```

```
"response":{"numFound":2,"start":0,"docs":[
  {
    "id":"6H500F0",
    "secret_sauce":2100.0,
    "sales_price":350.0,
    "why_score":{"
      "match":true,
      "value":1.052226,
      "description":"weight(features:cache in 2) [DefaultSimilarity], result of:",
      "details":[{"
```

### (7) debug

debug 参数可以指定多次，它支持以下可选的参数值：

- query：表示返回查询相关的调试信息；
- timing：表示返回查询的各个执行阶段的耗时信息；
- results：表示返回查询执行计划相关信息，类似于 MySQL 里的 SQL 执行计划。默认情况下，查询执行计划会返回大量的文本信息，为了提高可读性，使用了首行和 Tab 缩进的结构显示，此外可以指定额外的参数 debug.explain.structured = true，它会使得查询执行计划的输出信息按照 wt 参数设置的格式进行输出。
- all：表示返回所有跟当前查询请求相关的调试信息，与 debug = true 等效。

为了与 Solr 以前的旧版本保持兼容，debugQuery = true 可以使用新版本中的 debug = all 来代替。debug 参数如果未指定，那么默认不输出任何调试信息。

### (8) explainOther

explainOther 参数用于在一个主查询（即 q 参数指定的查询）内指定一个 Lucene 的简单查询，并对该查询执行查询计划。示例如下：

```
q=supervillians&debugQuery=on&explainOther=id:juggernaut
```

表示在主查询 `q = supervillians` 内对 `explainOther` 参数指定的 `id: juggernaut` 查询返回的每个索引文档执行查询计划。

如果此参数未指定，或者指定为空值，那么不返回任何调试信息。不配置默认为空值。

#### (9) timeAllowed

该参数用于指定查询执行的最大超时时间，单位为毫秒。若查询在限定的最大超时时间内未完成，那么会返回部分结果，但返回的 `numFound`（即结果集总数）、`facet count`（即 `facet` 查询统计的数量）、`result state`（即查询状态）等这些数据可能会不准确。

#### (10) segmentTerminateEarly

此参数是一个 `boolean` 值，如果设置为 `true`，并且 `solrconfig.xml` 中 `<mergePolicy-Factory/>` 配置为 `SortingMergePolicyFactory`，那么它会提前为每个段文件里的 `document` 进行排序，它使用的排序参数与主查询的 `sort` 参数兼容，然后 Solr 会尝试使用 `EarlyTerminatingSortingCollector` 结果收集器去收集最终返回的结果集。`EarlyTerminatingSortingCollector` 只会返回 `numCollected` 个索引文档即 `TopN` 个，所以如果段文件提前合并和排序了，可以避免段文件全部扫描，即提前中断段文件查询。此参数一般不建议开启，除非你在执行 `TopN` 查询时可能会用到。由于它会提前中断段文件查询，所以它与 `timeAllowed` 参数类似，最终返回的 `numFound`（即结果集总数）、`facet count`（即 `facet` 查询统计的数量）、`result state`（即查询状态）等这些数据可能会不准确。

#### (11) omitHeader

此参数是一个 `boolean` 值，如果它设置为 `true`，那么在最终返回的响应信息中将不包含响应头信息，比如查询耗时。若此参数未设置，那么默认值为 `false`。

#### (12) wt

`wt` 参数用于指定使用 `Response Writer` 来格式化查询返回的响应信息。默认采用 `JSON` 格式。

#### (13) cache

Solr 默认会对每个 `Query` 以及 `Filter Query` 返回的结果集进行缓存。如果想要禁用默认的缓存行为，那么你可以设置 `cache = false`。

你也可以设置 `cost` 选项来控制不缓存的 `Filter Query` 的执行顺序。`cost` 值越大表示它的执行开销越昂贵。通过 `cost` 选项你可以控制低开销的 `Filter Query` 先于高开销的 `Filter Query` 执行。

对于开销很高的 `Filter Query`，如果 `cache = false` 且 `cost >= 100` 且 `Query` 实现了 `PostFilter` 接口，那么 `Collector` 结果收集器等主查询和其他 `Filter Query` 全部执行完成后再执行该 `Filter Query`。示例如下：

```
fq={!frange l=10 u=100 cache=false cost=100}mul(popularity,price)
```

### (14) logParamsList

默认 Solr 会记录请求所有参数的日志，从 Solr 4.7 版本开始，你可以设置此参数来约束哪些请求参数应该记录日志，它可能会有助于你去控制日志只记录那些你认为比较重要的请求参数。

比如，你可以进行类似这样的定义：

```
logParamsList=q,fq
```

上面的示例表示只记录 q 和 fq 请求的参数到日志中。如果你想禁用日志记录功能，那么你可以设置 logParamsList 等于一个空值，比如 logParamsList = （注意，这里有一个空格）。

logParamsList 参数不仅仅可以应用于 Solr 查询请求中，其实 Solr 中的所有请求的日志记录都可以通过此参数控制。

### (15) echoParams

echoParams 参数用于控制在响应头信息中显示哪些请求参数。

表 5-4 解释了 echoParams 参数的各种设置的具体含义。

表 5-4 echoParams 参数设置及含义

示例	解 释
explicit	只有实际显式定义的请求参数以及下划线参数 _（它是一个 64 位的数字时间戳）才会在 response header 中返回。这也是默认值
all	显式定义的请求参数以及 Request Handler 类内部定义的默认参数以及 solrconfig.xml 中配置的默认参数以及下划线参数 _ 都会在 response header 中返回
none	不在 response header 中返回任何请求参数

下面是 response header 中默认包含的请求参数：

```
"responseHeader": {
  "status": 0,
  "QTime": 2,
  "params": {
    "fl": "bookName,[explain]",
    "indent": "true",
    "q": "bookName:head",
    "_": "1474825332836",
    "wt": "json"
  }
}
```

## 5.4 Lucene 的基本查询语法

Solr 的 Standard Query Parser 继承自 Lucene 的 Query Parser，因此你需要先了解 Lucene 中关于 Query Parser 支持的查询语法。Lucene 通过 Query Parser 提供了丰富的查询语法，内

部是使用 JavaCC 来实现查询关键字的语法解析。

在学习使用 Query Parser 之前，你需要先熟悉以下几点：

- ❑ 如果你通过编程的方式生成了查询表达式并将其传递给 Query Parser 解析，那么此时你应该认真考虑下是否应该直接通过 Lucene Query API 来构建你的 Query 对象，换句话说，设计 Query Parser 的初衷就是为了避免用户在查询时输入晦涩难懂的查询表达式，直接输入查询文本，然后通过 Query Parser 将其解析成查询表达式，你自己生成查询表达式再传给 Query Parser 就本末倒置了；
- ❑ 对于不需要分词的域，应该直接对其构造 Query，而不是将查询关键词传递给 Query Parser，因为 Query Parser 会借组 Analyzer 分词器对用户的查询关键词进行分词，既然你的域不需要分词还传递给 Query Parser 就有点多余；
- ❑ 在查询表单中，普通的查询文本需要传递给 Query Parser 进行解析，而对于日期范围、关键字等应该直接通过 Query API 构建 Query，而对于类似下拉列表的值不应该添加到查询关键字中，而应该直接通过选择的列表项的某个值构建 TermQuery。

一个查询语句由 Term 和操作符组成，而 Term 又分为两种类型：单个 Term 和 Phrase 短语。单个 Term 表示一个单词比如“test”“hello”。Phrase 短语是两头被双引号包裹的一组单词。多个 Term 可以通过 Boolean 操作符连接起来组成一个更复杂的 Query。



**注意** 分词器既可以用于索引创建，也可以用于对查询关键字中的 Term 和 Phrase 进行分词，所以当你选择使用一个分词器时需要清楚分词器在创建索引时是否会影响查询关键字的 Term 和短语的生成。换句话说，你在使用分词器时需要清楚划分 Index（索引时）和 Query（查询时）两个阶段，保证两个阶段互不干预，分别控制。

在构建查询时，你既可以指定一个在某个特定的域上进行查询，也可以不指定域在默认域上进行查询，你可以在任何域上进行查询，先指定域的名称，其后紧跟一个冒号，冒号之后就是你的查询 Term。举个例子，假定你的索引中包含了两个域：title 和 text，并且 text 是默认域，如果你希望查找索引文档中 title 为“The Right Way”且 text 包含“don't go this way”的文档，那么你可以输入：

```
title: "The Right Way" AND text:go
```

或者

```
title: "The Right Way" ANDgo
```

因为 text 是默认域，所以域名称可以省略不指定，如果你这样指定查询表达式，可能得不到你想要的结果：

```
title:the right way
```

对于上面的查询表达式，或许你的本意是想要查找 title 域中包含 the、title、way 其中任意一个关键字的索引文档，然而在 Lucene 查询语法中，空格默认会作为 OR 操作符来连接多个子查询表达式，即实际查询表达式会解析成 title: the or text:right or text:way。因为 right 和 way 前面没有指定域，所以使用了默认域。

Lucene 支持对单个 Term 进行通配符查询，对单个字符进行模糊那就使用“?”通配符，对于多个字符进行模糊，那么就使用“\*”通配符，单个字符的通配符查询你可以这样执行查询，比如你想要查询“test”或“text”：

```
te?t
```

对多个字符进行模糊，你可以这样执行查询：

```
test* 或者 t*t
```

但是你不能将“?”或“\*”通配符放置在查询表达式的开头，因为这种前缀模糊查询性能非常烂，为了防止用户使用，默认已经被禁止使用了。如果你执意想要实现类似关系型数据库的 like %xx 之类的模糊查询，那么你可以借助 NGram 实现，比如“book”你可以 NGram 分词分成 b bo boo book，这样你输入 b bo boo book 都能搜到，但分词粒度太细会增大索引体积，同时也会匹配到很多不相关的文档。

Lucene 还支持基于 Levenshtein Distance 实现的 Fuzzy Query，你可以通过在单个 Term 后面添加“~”符号来指定使用 Fuzzy Query。比如你想要使用 Fuzzy Query 返回一个跟“roam”拼写相似的单词，那么你可以这样指定查询表达式：

```
roam~
```

这个查询会返回包含 foam、roams 等单词的索引文档。当然你也可以在“~”符号后面指定一个相似度数来限定返回的结果，相似度数取值范围为 [0 ~ 1]，默认值为 0.5。示例如下：

```
roam~0.8
```

Lucene 支持 Proximity Searches (即邻近查询)，表示查找两个单词之间间隔指定距离的索引文档，使用邻近查询你可以在 Phrase 短语之后添加一个“~”符号，比如查询“apache”和“jakarta”之间间隔 10 个单词的索引文档你可以这样指定查询表达式：

```
"jakarta apache"~10
```

Lucene 的 Range Query (范围查询) 允许匹配指定域的域值在特定区间范围的索引文档，使用中括号“[]”表示包含边界，使用花括号“{}”表示排除边界。默认 Lucene 中不支持 Date 域，所以在 Lucene 中时间日期类型数据你只能使用字符串或者转成 Long 类型的毫秒数，然后你才可以使用 Range Query。范围查询离不开大小比较，默认大小比较标准是按照字符的 ASCII 码值进行比较的。下面是一个 Range Query 的查询表达式示例：



```
mod_date:[20020101 TO 20030101]
```

这个查询表达式表示查询 `mod_date` 域的域值在 20020101 和 20030101 之间的所有索引文档。

Lucene 还提供了对匹配文档中包含指定 Term 的相关性级别即 Term 的权重值，权重值用数字表示，Term 的相关性越高那么它的 Boost 权重值就应该越大。为一个 Term 指定权重值可以使用 ^ 符号并在它后面添加一个权重数值。通过为一个 Term 添加权重值可以方便用户控制返回的索引文档的相关性。比如你想要查询 “jakarta apache”，并且你想要 “jakarta” 在文档中拥有更高的权重，那你可以在 jakarta 这个 Term 后面指定权重值。你可以这样指定：

```
jakarta^4 apache
```

这使得包含 “Jakarta” 这个 Term 的索引文档具有更高的相关性，你也可以对短语指定权重，比如：

```
"jakarta apache"^4 "Apache Lucene"
```

默认权重值为 1，尽管权重值必须是正数，但它可以被指定为小数，比如：

```
jakarta^0.2 apache
```

Lucene 的查询表达式还支持 Boolean 操作符，它允许多个 Term 之间通过 Boolean 逻辑操作符进行连接。Lucene 支持的 Boolean 操作符有 AND、“+”、OR、NOT、“-”。注意：所有的 Boolean 操作符必须大写。

Lucene 默认使用的是 OR 操作符，这意味着如果你没有显式指定 Boolean 操作符，那么在两个 Term 之间默认使用 OR 操作符进行连接。OR 操作符连接两个 Term 并匹配包含两个 Term 中任意一个的索引文档，这类似于求两个集合的并集。比如查询一个文档包含 “jakarta apache” 或者仅仅只包含 “jakarta”，那么你可以这样指定查询表达式：

```
"jakarta apache" jakarta 或者 "jakarta apache" OR jakarta
```

AND 操作符用于匹配同时包含的两个 Term 的索引文档，这类似于求两个集合的交集。比如查询一个文档同时包含 “jakarta apache” 和 “Apache Lucene”，你可以这样查询：

```
"jakarta apache" AND "Apache Lucene"
```

“+” 加号操作符用于匹配文档中必须包含指定的 Term，它必须指定在 Term 的前面。比如查询一个文档必须包含 “jakarta” 并且可能包含 “lucene”，你可以这样查询：

```
+jakarta lucene
```

NOT 操作符用于排除包含指定 Term 的索引文档，比如查询一个文档包含 “jakarta apache” 但不能包含 “Apache Lucene”，你可以这样查询：



```
"jakarta apache" NOT "Apache Lucene"
```

同理还有“-”减号操作符，它与 NOT 作用类似。

Lucene 支持使用小圆括号“()”对查询表达式进行分组形成多个子查询表达式，当你想要控制 Boolean 查询的逻辑时，它可能会比较有用。比如查询文档包含“jakarta”或者“apache”并且必须包含“website”，你可以这样查询：

```
(jakarta OR apache) AND website
```

Lucene 支持对单个域的多个查询表达式使用圆括号“()”进行分组，比如查询文档中包含“return”并且包含短语“pink panther”，那么你可以这样查询：

```
title:(+return +"pink panther")
```

另外需要引起你注意的是，Lucene 中以下字符需要进行转义：

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \
```

因为它们在 Lucene 的查询表达式中拥有特殊的含义，所以如果想要在查询表达式中使用这些字符，那么你需要对其进行转移，只需要在上述特殊字符的前面添加斜杠字符“\”即可，比如你想要查询的文档中包含“(1+1=2)”，那么你可以这样查询：

```
\(1\+1\=2)
```

## 5.5 Solr 的标准查询语法解析器

Solr Standard Query Parser 是 Solr 的标准查询解析器，也是 Solr 默认使用的查询解析器，它继承自 Lucene 的 Query Parser，因此 Lucene 的查询表达式语法全部适用于 Solr。但 Solr 的标准查询解析器与 Lucene 的查询解析器还是有一些区别的，即两者并不是完全对等的。

Solr 标准查询解析器支持如表 5-5 所示的参数。

表 5-5 Solr 标准查询解析器支持的参数

示例	解 释
q	根据 q 参数值构建一个查询，q 参数值必须符合 Solr 标准查询解析器支持的语法，此参数是必须指定的
q.op	用于指定 boolean 操作符，可选值有 AND/OR，默认值为 OR。你可以在 schema.xml 中配置默认的 boolean 操作符
df	用于指定查询的默认域，当查询文本未指定查询域，那么会使用默认域，你同样可以在 schema.xml 中配置默认域

Solr 标准查询解析器与 Lucene 查询解析器之间的区别：

□ 通配符查询中“\*”星号可以用在范围区间的两端，比如：

- field: [\* TO 100]: 查询域值小于等于 100 的索引文档;
- field: [100 TO \*]: 查询域值大于等于 100 的索引文档;
- field: [\* TO \*]: 查询返回所有索引文档;
- -field: [\* TO \*]: 查询域值为空的索引文档。

❑ 支持 filter() 语法将一个普通查询的结果集缓存到 Filter Query 的缓存区, 比如 `q = features: songs OR filter(inStock: true)`。

❑ 在 Solr 中区间范围查询、前缀查询、通配符查询都是常量评分查询, 即返回的每个索引文档的评分都是相同的, 评分因素 TF、IDF、索引文档的权重、coord 都会被忽略。

❑ TrieDateField 域的区间范围查询中支持的日期格式不同, 在 Solr 中支持 UTC 日期格式, UTC 日期时间格式为 YYYY-MM-DDThh:mm:ssZ。比如:

- timestamp: [\* TO NOW]: 返回从过去截止到目前的所有索引文档;
- createdate: [1976-03-06T23: 59: 59.999Z TO \*]: 返回从 1976-03-06T23: 59: 59.999Z 之后的所有索引文档;
- createdate: [1995-12-31T23: 59: 59.999Z TO 2007-03-06T00: 00: 00Z]: 返回两个时间区间内的索引文档;
- pubdate: [NOW-1YEAR/DAY TO NOW/DAY + 1DAY]: 返回从去年的今天到明天之间的所有索引文档。

## 5.6 Solr DisMax

DisMax Query Parser 设计的初衷是处理用户输入的简单查询短语 (不包括复杂的语法) 并且基于每个域的重要性为跨多个域的个别 Term 添加不同的权重。提供的一些额外参数使用户能够基于特定的规则去影响评分 (不依赖于用户的输入)。

一般来说, DisMax Query Parser 接口看起来更像是 Google 而不像标准的 Solr 请求处理器接口。这种相似性使得 DisMax 成为能适用于多种应用程序的 Query Parser, 它接收一个简单的语法, 并且几乎不会产生错误信息。也就是说 DisMax 的查询语法是比较松散的, 即便语法不正确, 只会不返回结果并不会返回错误信息给客户端用户。

DisMax 支持极其简单的语法, 它是 Lucene QueryParser 语法的一个子集。因为在 Lucene 中, 引号字符是用来组织 Phrase 短语, “+” 加号用来表示强制要求必须满足, 而 “-” 减号用来表示可选的条件。Lucene Query Parser 里的其他所有特殊字符 (AND 和 OR 除外) 都会被转义以简化用户的使用体验。DisMax 负责根据用户可能使用了 Boolean 条件的输入构建一个跨多个域的 DisMax Query 并指定权重, 同时它还还为 Solr 管理者能提供一个额外的 Boosting Query、Boosting Function 以及 Filter Query, 以便于能够人工干预查询结果。DisMax 的所有参数在定义于 solrconfig.xml 配置文件中的 Request Handler 里都赋予了默认值, 你可以通过将参数拼接到 Solr 的查询 URL 后面进行默认值覆盖。

DisMax 名称背后涵盖了什么技术概念呢？DisMax 表示 Maximum Disjunction。下面是关于“Maximum Disjunction”和“DisMax Query”概念定义的解释：

假如你输入的查询关键字是“solr”，那么 DisMax 会将其解析为 fieldA: solr^2, fieldB: solr^1.2，这样相当于此时有 2 个子查询，如果是简单的 Boolean 查询，那么此时会求 2 个子查询得分的总和作为最终的得分，而在 DisjunctionMaxQuery 中，则是取 2 个子查询得分的最高得分作为最终的得分，而有可能两个文档的最终得分相同，这时候为了加大某个文档的权重而引入了 tie breaking 参数（tie 参数不指定，默认值是 0.0），计算公式也变为：（最高分）+（tie 参数）\*（其他匹配条件的得分）。假如查询在 A、B 两个域上都有 term 命中，它们的得分可能是如表 5-6 所示的。

表 5-6 A、B 域的得分示例

	A	B	最高分
文档 1	0.5	0.8	0.8
文档 2	0.8	0.1	0.8

此时两个文档的得分是相同的，但显然这不合理，因为文档 1 中 A、B 两个域的得分明显要高，所以文档 1 的得分应该高些，所以引入 tie 参数，再根据上面计算公式重新计算：

文档 1 的得分 = 0.8 + 0.1 \* 0.5 = 0.85；

文档 2 的得分 = 0.8 + 0.1 \* 0.1 = 0.81。

从而使得文档 1 的得分高于文档 2，达到预想的结果。这就是所谓的“DisMax”。

不管你理不理解上面的解释，请记住 DisMax 设计的目的就是提高易用性以及能够接收任何输入但不返回任何错误信息，提高用户的使用体验。

除了通用的请求参数、高亮参数和 facet 参数之外，DisMax 查询还支持如表 5-7 所示的参数。

表 5-7 DisMax 支持的参数

参数	描 述
q	即 query 的缩写，用于指定查询输入的原始字符串
q.alt	当 q 参数未指定时，会调用标准查询解析器来解析当前参数值并构造一个 Query
qf	即 query field 的缩写，用于指定在哪个域上执行查询，若此参数未指定，那么默认取 df 参数， 示例： qf = "fieldOne^2.3 fieldTwo fieldThree^0.4"
mm	用于指定应该匹配的条件的最小数目
pf	即 phrase fields 的缩写，用于指定一组域名称来构造 Phrase Query（短语查询）
ps	即 phrase slop 的缩写，用于指定 pf 参数构造的短语查询需要用到的编辑距
qs	即 query phrase slop 的缩写，用于指定 q 参数构造的短语查询需要用到的编辑距
tie	用于 DisjunctionMaxQuery 中的一个 Float 类型的决定性因素，取值范围 [0, 1]，当 tie = 1，那就是取每个子条件查询的总评分作为最后得分，当 tie = 0，那就是取每个子条件查询的最大评分作为最后得分，通常 tie 参数设置为 0.1 更有用

(续)

参数	描 述
bq	即 boost query 的缩写，用于构造一个 Boost Query，当你希望对某部分索引文档加权，比如当你希望最近上架的书应该靠前显示时，bq 参数会很有用，示例： q = cheese&bq = date: [NOW/DAY-1YEAR TO NOW/DAY]
bf	即 boost function 的缩写，用于根据 bf 参数构造一个 Function Query，从而干预主查询返回文档的评分。示例： recip (rord (myfield), 1, 2, 3)^1.5

(1) mm

默认通过 q 参数指定的查询条件都是“Should”，在 Lucene 中 Boolean 查询的条件满足有 3 种类型：MUST（必须满足）、MUST NOT（必须不满足）、SHOULD（应该满足）。当一个查询包含了这 3 种类型的查询条件，那么此时你可以通过指定 mm 参数来要求查询至少必须匹配几个查询条件。指定 mm 参数有如表 5-8 所示几种可选方式。

表 5-8 指定 mm 参数的方式

参数	示 例	描 述
正整数	3	表示不管主查询中有多少个查询条件，必须满足至少 3 个查询条件的索引文档才会被返回
负整数	-2	表示至多有 2 个查询条件不满足的索引文档才会被返回
百分比	75%	表示至少有 75% 个查询条件满足的索引文档才会被返回
负数百分比	-25%	表示至多有 25% 的查询条件不满足查询条件你的索引文档才会被返回
整数 + < + 百分比或 正整数 + < 负整数	2<-1 5<80%	如果查询条件小于 3，那么会要求所有查询条件必须全部满足； 如果查询条件为 3 到 5 个，那么至多一个不匹配； 如果查询条件 >5，那么会要求至少满足 80% 个查询条件

当你指定了 mm 参数，你需要注意以下几点：

- ❑ 当处理百分比时，负数可以用来处理一些边界情况。比如 75% 和 -25% 在只有 4 个查询条件时表示同样的含义，但是当有 5 个查询条件时，75% 表示至少有 3 个查询条件必须满足，而 -25% 则表示至多有一个查询条件不满足，即至少有 4 个查询条件必须满足；
- ❑ 当根据 mm 指定的参数计算后发现不需要任何一个查询条件必须满足，那么此时 Boolean 查询会隐含的强制要求至少有 1 个查询条件必须满足；
- ❑ 无论根据 mm 参数计算后的数值是多少，即便它大于查询条件总数，或者它小于 1，最终匹配查询条件的个数的取值范围都必须在 [1，查询条件总数] 这个区间之内；
- ❑ 当一个查询跨了多个域，并且每个域都配置了使用不同的分词器，那么每个域对应的查询条件的个数可能会不同，在这种情况下，mm 参数会取两者的最大值，假如一个域的查询条件被判定为停用词，那么这个域的查询条件个数会小于其他域，并且该域上的查询也不会有匹配，即便 mm 参数设置为 100%；

□ mm 参数默认值是 100%，表示默认所有查询条件必须匹配。

## (2) pf

此参数用于构造一个 Phrase Query (短语查询)，当 q 或 fq 参数指定了，那么会根据 q 或 fq 参数在 pf 参数指定的域上构造 Phrase Query，那么对于那些匹配的文档中 term 更靠近的文档的评分会被加权。举个例子，假如 q = “java solr” 表示构造一个 Phrase Query，pf = fieldA, fieldB 表示根据 q 参数来 fieldA 和 fieldB 两个域上分别构造 Phrase Query，那么两个域上的短语查询得分越高的说明 java 和 solr 这两个 term 在原始索引文档中离得越近，那么那些匹配的 Term 离的更近的文档应该被加权从而评分更高，评分较高的自然就靠前显示。

## (3) ps

即 phrase slop 的缩写，用于指定 pf 参数构造的短语查询需要用到的编辑距，常常和 pf 参数搭配使用。

## (4) qs

即 query phrase slop 的缩写，用于指定 q 参数构造的短语查询需要用到的编辑距，跟 ps 参数类似，但 qs 参数是用于为 q 参数构造的短语查询设置编辑距。举个例子，可能你的 q = “java solr”，因为被双引号包裹起来默认就会被解析为短语查询。假如 qs = 2，那么此时 q 参数就等价于 “java solr” ~2。也就是说仅当 q 参数被解析为短语查询时设置 qs 参数才有意义。

## (5) bf

即 boost function 的缩写，用于根据 bf 参数构造一个 Function Query，从而干预主查询返回文档的评分。bf 参数其实只是 bq 参数与 {!func} 的一个简写形式，比如你想要查询最近添加的索引文档，你可以这样指定查询语法：

```
bf=recip(rord(creationDate),1,1000,1000)
```

或者

```
bq={!func}recip(rord(creationDate),1,1000,1000)
```

## 5.7 Solr eDisMax

eDisMax 即 Extended DisMax，它是 DisMax Query Parser 的升级版，它除了支持 Lucene Query Parser 的所有参数、DisMax Query Parser 的所有参数之外，它还支持：

□ 在 Lucene 语法模式下，将 and、or 解析为 AND、OR；

□ 支持一种 “magic field” (魔域) 比如 \_val\_ 和 \_query\_。魔域并不在 schema.xml 中真实存在，但是你可以使用它做一些一些特殊的事情，比如使用 \_val\_ 实现 function query，使用 \_query\_ 实现 nested query。如果 \_val\_ 域用于 term query 或 phrase query，那么

它的值会被解析成一个 Function;

- 包括改进了部分智能的特殊字符转义以防止语法错误, 同时仍然支持 FieldQuery, Phrase Query 以及 “+” / “-” 加减号操作符;

- 增强了对 Span Query 的支持, 在对拥有邻近 Term 的文档加权之前, 你不需要在文档中匹配所有的词;

- 包括高级停用词处理;

- 增强了对 Boost Function 的支持, 在 eDisMax 模式下, DisMax 的 bf 和 bq 参数依然支持;

- 支持纯否定的 nested query (嵌套查询): 比如支持 + foo (-foo) 查询, 它会匹配所有索引文档, 它能够限制终端用户查询哪些域或者拒绝直接的域查询。

除了能够安全地处理用户输入的查询文本以及随心所欲的解释查询语法, eDisMax 最有用的特性就是支持跨多个域查询, 而不是强制的将查询文本复制到一个默认域上进行查询。eDisMax 会自动将查询文本的每部分应用到指定的多个域上进行查询。在 Lucene 中, 你想根据查询文本 “solr in action” 在多个域上构造一个查询可能需要这样写:

```
((title:solr) OR (description:solr) OR (author:solr)) AND ((title:in) OR
(description:in) OR (author:in)) AND ((title:action) OR (description:action)
OR (author:action)))
```

即分别在 title、author 和 description 这个 3 个域上执行查询。相比之下, 使用 eDisMax 查询解析器来实现显得更轻松容易, 比如这样:

```
q=solr in action&qf=title description author
```

使用 eDisMax 查询解析器来实现, 会使你的查询文本看起来更紧凑, 而不是被拆分为一个个词然后分散在多个查询域上。同时它还能够将你的 IDF 统计从每个查询域分离出去, 这有助于文档的相关性评分。使用 eDisMax 查询解析器的另外一个好处就是将多个查询域通过 qf 参数组织在一起, 并且你能够集中对每个域进行权重设置, 比如:

```
q=solr in action&qf=title^1.5 description author^3
```

eDisMax 其中一个强大功能就是能纯天然的支持对比较靠近的 term 自动加权, 而不像传统的 Lucene Query Parser 默认需要按照短语相等来匹配, 而不管两个 Term 中间相隔的有多近。eDisMax 另外一个功能就是它能对独立主查询的 Function 应用任意的相关性权重。

对于用户输入的查询文本字符串中的每个词来说, DisMax 会依据该词以及 qf 参数指定的多个域构建一个 DisjunctionMaxQuery 查询。然后 DisjunctionMaxQuery 会被放进一个 Boolean Query 中并带上 mm 参数的最小布尔查询条件数目限制。如果还指定了其他参数, 那么会在当前 Boolean Query 外层再包装一个更大的 Boolean Query, 其他参数比如 bf, bq, pf, pf2, pf3, 'ps2', 'ps3' 会被当作可选的查询条件。唯一一个比较复杂的查询条件来自于



pf 参数, pf 参数会被构造成 DisjunctionMaxQuery 查询, 它包含了针对 pf 参数指定的每个域的整个查询短语。

eDisMax 除了支持 DisMax 支持的全部查询参数之外, 还支持以下查询参数:

#### (1) pf/pf2/pf3

pf 参数可以用于对邻近 Term 的索引文档的评分进行加权, pf 参数与 qf 参数使用相同的格式, 你可以指定多个域, 同时可以为每个域单独指定权重, 多个域之间使用逗号分隔, eDisMax 会尝试对 q 参数中指定的查询文本中包含的所有 term 构造一个 phrase query, 如果在 pf 参数指定的域上能够精确匹配到索引文档, 那么就会将指定的权重应用到匹配到的索引文档上。

除了 pf 参数之外, eDisMax 查询解析器还支持 pf2、pf3 参数, 这些参数的功能与 pf 参数类似, 但是 pf2、pf3 参数并不需要匹配 q 参数中包含的所有 term, pf2 参数需要匹配两个词组成的 gram, 而 pf3 参数只需要匹配 3 个词组成的 gram 即可。举例说明, 假如用户查询 “solr finds relevant documents”, 那么对于 pf2 参数, 如果索引文档中包含 “solr finds”、“finds relevant”、“relevant documents” 其中任意一种情况, 那么该索引文档就会被加权。同理, 对于 pf3 参数, 如果索引文档中包含 “solr finds relevant”、“finds relevant documents” 其中任意一种情况, 那么该索引文档就会被加权。当然, 如果你还考虑设置 Phrase Query (短语查询) 中的 slop (编辑距) 问题, 那么两个 term 之间间隔 slop 限定范围内的 term 会被视为邻近 term, 那么该索引文档也会被加权。

#### (2) ps/ps2/ps3

当你使用 pf 参数时, 你可能不希望查询中所有 term 都精确匹配, 你可以使用 ps (即 phrase slop 的缩写) 参数, 这样只要两个 term 之间间隔的其他 term 的个数在 phrase slop 限定的范围内, 那么就认为该索引文档匹配查询条件应该被返回。eDisMax 查询解析器还支持 ps2、ps3 参数, 它允许覆盖默认的 ps 参数值, 当 ps2 和 ps3 参数未指定时, 默认会使用 ps 参数值。ps2 与 pf2 参数相对应, ps3 与 pf3 参数相对应。也就是说 ps 参数需要跟 pf 参数搭配使用才有意义。

#### (3) tie

当某个查询 Term 在索引文档的多个域上都匹配时, 可能会发生两个索引文档的最终得分是相同的, 但有可能两个文档分别在每个域上的评分不一致, 因为最终得分是计算每个域上得分的最高分作为文档的最后得分。这显然不公平, 好比两个歌手参加歌唱比赛, 评委给出的最高分都是 9 分 (假定满分 10 分), 假如以最高分为最终成绩, 此时可能两个人成绩相同, 分不出高下, 但是如果对于 A 选手每个评委打的分数依次是 0.9、0.8、0.9、0.8、0.9, 而对于 B 选手每个评委打的分数依次是 0.8、0.6、0.5、0.8、0.9, 显然 A 选手成绩更好。Solr 中同理, 为了让两个得分相同的索引文档能够区分开, 引入了 tie 参数, 最终计算公式为:

最高分 + tie 参数值 \* (其他域的评分)



`tie` 参数决定了除了最高分的那个域应该作为整体相关性得分之外，其他域上的相关性得分也应该作为整体得分的参考因素。`tie` 参数值默认值为 0.0，表示除了最高分所在域贡献得分之外，其他域不贡献任何相关性得分，即完全按照所有域上得分的最高分作为文档的最后得分。而假如 `tie` 参数设置为 1.0，则表示所有域都贡献相关性得分，即相当于求所有域上得分的总和了，这也是 Lucene 中的查询解析器的处理方式。此时索引文档评分计算就是求和而不是求最大值啦。

#### (4) Field aliasing

Field aliasing 即域别名，你可以为查询中的域指定一个别名，即你可以为用户指定一个本地化的域名称或者一个更容易记住的别名，但它们并不在 `schema.xml` 中真实存在。示例如下：

```
f.myalias.qf=realfield      // 为 realfield 域定义一个别名为 myalias
myalias:foo                 // 用户输入 myalias:foo 字符串进行查询
realfield:foo               // 实际执行的是 realfield:foo
```

不仅可以为单个域指定别名，你还可以为多个域指定一个别名，示例如下：

```
// 定义别名
&f.who.qf=name^5.0namealias^2.0&f.where.qf=address^1.0city^10.0state
// 用户实际查询输入
who:foo
```

当用户输入 `who: foo` 其实就等价于在 `name` 和 `namealias` 这两个域上面执行查询，同时由于在定义别名时，我们还为每个域单独设置了权重，而这对于用户来说都是透明不可见的。但此时用户其实还是可以直接对真实域进行查询的。可能有时候你并不想把 `schema.xml` 中的真实域名称暴露给用户，你想直接禁止用户通过 `schema.xml` 中的真实域进行查询，而是通过定义的域别名来进行查询，那么此时你需要 `uf` 参数，`uf` 即 `user field` 的缩写也是用户域，所谓用户域即面向用户的域，并不是真实存在的域，通常是域的一个别名。示例如下：

```
&uf=who,where
```

`who` 和 `where` 就是我们在上个示例中定义的两个别名，这样用户就只能通过这两个别名来进行查询，对于用户来讲，`who` 和 `where` 就相当于两个“伪域”。而且一定程度上也简化了用户查询，因为我们可以为多个域起一个别名，对于用户来说，感觉像是在一个域上执行查询，而且你同时可以为每个域指定权重，而这些操作都隐含在别名定义内部，用户不用再关心每个域的权重。示例如下：

```
/select?defType=edismax&
f.who.qf=personLastName^30 personFirstName^10&
f.what.qf=itemName company^5&
f.where.qf=city^10 state^20 country^35 postalCode^30&
q=...
```

其中 `q` 参数才是用户在查询时需要指定的, 此时用户只需要类似这样输入即可:

```
q=who:(trey grainger) what:(solr) where:(decatur, ga)
```

`uf` 参数的默认值是 `uf = *`, 表示用户可以通过这样的语法 `fieldName: expression` 对 `schema.xml` 中的任意域执行查询, 如果想限制用户只能对某一个域执行查询, 那么你可以这样设置:

```
uf=title // 表示用户只能对 title 这一个域执行查询
```

当然也可以限制用户只能对指定的几个域执行查询, 多个域之间使用空格分割, 此时你可以这样设置:

```
uf=title city pubDate // 表示用户只能对 title city pubDate 这 3 个域执行查询
```

如果想禁止用户对 `schema.xml` 中的任何域执行查询, 那么你可以这样设置:

```
uf=-* // 使用负号进行取反, 表示禁止用户对任何域执行查询
```

如果你想限制用户访问的域只有 2 ~ 3 个, 而实际域的总个数几十个, 并且一个使用空格分割太麻烦, 此时你可以这样设置:

```
// 表示只对用户禁用 hiddenField1 和 hiddenField2 这 2 个域, 其他域都允许
uf=* -hiddenField1 -hiddenField2
```

`uf` 参数除了能接收域名之外, 还可以接收域的别名, 当你想要全面控制用户的查询, 你可以将 `uf` 参数与域别名结合起来使用, 这样你就可以限制用户只能在定义的域别名上执行查询, 将实际存在的域与用户完全隔离, 比如你可以这样设置:

```
/select?defType=edismax&
&df=text&
f.who.qf=lastName^30 firstName^10&
f.what.qf=itemName companyName^5&
uf=who what&
q=+who:(timothy potter) +what:(solr in action) +"big data"
```

`uf = who what` 表示的含义是用户只能在指定的 `who` 和 `what` 这两个域别名上执行查询, 此外, 用户还可以在默认域上执行查询, 默认域为 `text`, 当默认域的值你可以通过 `df` 参数进行重置或者在 `slorconfig.xml` 中进行默认域配置。

## (5) mm

我们知道, 在 Lucene 中, AND 和 OR 表示必须匹配和应该匹配, 查询表达式 “hello AND world” 可以被重写为 “+ hello + world”, 表示 `hello` 和 `world` 这两个 term 都必须匹配, 查询 “big OR brown OR cow” 表示 `big brown cow` 这 3 个 term 至少要匹配其中任意一个。通常你可能希望至少有一个查询条件匹配, 而不想关心具体是哪个查询条件匹配了。而传统的 Boolean 查询需要标识出每个可能的查询条件并用 Boolean 操作符去连接, `eDisMax` 查询

解析器通过 mm 参数模糊了传统 Boolean 查询的逻辑。mm 参数允许你定义一个数字或百分比，该数字或百分比表示 term 在索引文档中匹配的最小个数或最小百分比。只有达到限定的匹配个数或比例后，该索引文档才会被判定为匹配查询条件并返回给用户。利用 mm 参数你可以有效操控你的搜索程序的正确率（Precision）和召回率（Recall），因为它并没有要求所有 term 都必须匹配，仅仅只要求部分匹配。

举个例子，假如你有这样一个查询：

```
/select?q={!edismax mm="2<50% 4<-45%" v=$example}&example=...
```


基于上面的查询，对于 example 参数给定不同的参数将表示不同的含义，如表 5-9 所示。

表 5-9 example 参数含义表

example = solr	所有 term 必须匹配
example = solr is	所有 term 必须匹配
example = solr is a	至少一个 term 匹配（实际 33.333%，四舍五入到 50%）
example = solr is a search	至少 2 个 term 匹配（直接匹配率等于 50%）
example = solr is a search engine	至少 2 个 term 匹配（实际匹配率 40%，四舍五入到 55%）

(6) mm.autoRelax

如果此参数设置为 true，那么当某个查询条件被停用词过滤器给剔除掉之后，mm 参数的约束会自动变得不那么严格了。当你的查询条件被停用词过滤器移除掉之后导致没有命中任何索引文档时，你可以设置此参数来作为解决此类问题的解决方案。

 **注意** 开启此参数可能会导致一些你不希望看到的副作用，比如会损失一些查询精度，但这一切取决于你索引数据的性质。

(7) bf


用户可以使用 Solr 提供的 Function(函数) 作用于指定的域上，从而来干预文档的评分，在使用 Function 的同时还可以指定权重值，比如：recip (rord (myfield), 1, 2, 3)^1.5。

bf 参数语法：

```
_val_: "...function..."
```

比如

```
_val_: "recip(ms (NOW, mydatefield) "
```

 **注意** bf 参数可以指定多次。

### (8) lowercaseOperators

这是一个 boolean 参数, 用于指定小写形式的 “and” 和 “or” 是否应该被视为 Solr 中的 Boolean 逻辑操作符 “AND” 和 “OR”。

### (9) stopwords

这是一个 Boolean 参数, 用于指定是否将分词器的 StopFilterFactory 应用于用户查询时输入的查询字符串, 即是否对用户查询时输入的查询字符串进行停用词过滤。如果配置为 false, 那么 StopFilterFactory 在查询阶段将会被忽略。

eDisMax 查询解析器全方位支持 Lucene 查询解析器提供的查询语法, 同时还提供了很多额外的特性, 比如跨多个域查询、清洗用户的输入、域别名、限制用户能够查询哪些域、应用多种查询修饰符来提升短语的相关度、其他权重因子等。因为 eDisMax 查询解析器是 Lucene 查询解析器的一个超集, 所以你可能会想, 既然如此, 那还有谁会去使用 Lucene 的查询解析器呢? 对于一个典型的面向用户的应用程序来说, 通常会使用 eDisMax 查询解析器, 因为在应用层再重新实现一个类似 eDisMax 查询解析器提供的对用户友好的查询功能是毫无意义的, 除非你确实有一些特殊需求。eDisMax 查询解析器可以为应用程序生成所有的 Solr 查询, 然而, 你可能更喜欢使用 Lucene 查询解析器, 因为或许根本用不着 eDisMax 查询解析器提供的额外的强大特性。

当然 eDisMax 查询解析器也有一些不足, 首先就是需要去熟悉 eDisMax 查询解析器是如何处理跨多个域的查询的。如果把所有 term 都放在单一的域上执行查询, 肯定比使用 eDisMax 查询解析器跨多个域执行查询速度要快些。但这并不是 Lucene 查询解析器的优势, 因为你仍然可以使用 eDisMax 查询解析器在单一的域上执行查询。

eDisMax 查询解析器会给基于 Solr 的应用程序带来额外的代价, 这也是人们所诟病的。eDisMax 查询解析器的一个典型特征就是相关性评分, 不像 Lucene 查询解析器只关心 q 参数包含的所有 term 的相关性, 而不关心在哪个域上查询。eDisMax 查询解析器默认只计算最高分域的相关性, 比如:

```
/select?q={!edismax qf=title content}solr
/select?q=title:solr OR content:solr
```

理论上讲, 你可能预期它们返回的文档的相关性评分是相同的, 因为在 eDisMax 查询中, 你要求 Solr 在 title 和 content 两个域上执行查询, 然而内部 eDisMax 实际上只使用最高分的那个域的相关性评分。因此, 如果 “solr” 这个关键词的相关性评分在 title 域上更高的话, 那么最终文档的相关性评分只与 title 域有关了, 而 content 域的相关性评分就被忽略了。然而, Lucene 查询解析器不同, 它会考虑每个域的相关性评分, 并最终对每个域的相关性评分进行求和, 而 eDisMax 查询解析器是求每个域的相关性评分的最大值。也就是说 eDisMax 查询解析器考虑的是指定 term 在每个文档的哪个域上匹配度最高, 而 Lucene 查询解析器考虑的是文档在每个域上的总得分, 就好比学生考试, 老师评比哪个学生优秀,

对于 eDisMax 查询解析器来说,老师看重的是学生在哪个学科上更突出,比如该学生语文 60 分,而数学考了 149 分,那么该学生总分按数学最高分 149 计算,而对于 Lucene 查询解析器来说,老师看重的就是学生各学科的总分即综合实力而不是某一科的实力。

其实 eDisMax 查询解析器允许你通过使用 tie 参数来改变默认行为。如果 tie = 0.0, 那么就是默认行为即求每个域上得分的最大值;如果 tie = 1.0, 那么就跟 Lucene 查询解析器的默认行为一致;如果 tie 参数设置为 [0, 1] 之间的数值,那么就能同时兼顾两种情况,既考虑最高分优势又考虑了每个域上的综合因素。

对于大部分基于 Solr 的应用程序来说,可能并不是很关心某个关键词在每个域上得分的不同,那么此时对于那些刚接触 Solr 搜索的新手来说,建议使用 eDisMax 查询解析器。如果你的搜索程序对于相关性评分要求很挑剔很严格,而且你也很关心内部的评分细节的话,那么可能需要寻找一种更好的方式来提升查询的相关性评分。

## 5.8 Solr 的其他查询语法解析器

除了我们之前讨论的几种主要 Query parser 之外, Solr 还提供了很多种其他 Query Parser 如图 5-2 所示,它们可以用于实现一些特殊需求。这一节将具体讲解其他 Query Parser 并举例说明如何使用它们。

Block Join Query Parsers	Boost Query Parser	Collapsing Query Parser
Complex Phrase Query Parser	Field Query Parser	Function Query Parser
Function Range Query Parser	Graph Query Parser	Join Query Parser
Lucene Query Parser	Max Score Query Parser	More Like This Query Parser
Nested Query Parser	Old Lucene Query Parser	Prefix Query Parser
Raw Query Parser	Re-Ranking Query Parser	Simple Query Parser
Spatial Query Parsers	Surround Query Parser	Switch Query Parser
Term Query Parser	Terms Query Parser	XML Query Parser
DisMax Query Parser	Extended DisMax Query Parser	

图 5-2 Solr 内置 Query Parser

### (1) Field Query Parser

Field Query Parser 用于针对用户输入的查询文本在指定的域上构造一个 TermQuery 或者 Phrase Query (短语查询)。其中的 f 参数表示构造的 TermQuery 或者 PhraseQuery 应该在哪个域上进行查询,使用语法如下所示:

```
// 表示对用户输入的 "hello world" 在 myfield 域上构建一个 PhraseQuery
{!field f=myfield}hello world
```

### (2) Term VS Raw Query Parser

Term Query Parser 可以用于直接在 Solr 索引中查找值,与 Field Query Parser 相比, Term Query Parser 可以用于过滤 Facet 或 Terms 组件的返回值。它的语法如下:

```
{!term f=mystemmedtextfield}engin
```

```
{!term f=mystringfield}Single Term with Spaces
{!term f=myintfield}1.5
```

与 Field Query Parser 一样，f 参数也是表示在哪个域上执行查询。Solr 也包含了一个相似的实现：Raw Query Parser。Term Query Parser 与 Raw Query Parser 之间的唯一区别就是 Raw Query Parser 是直接针对 Solr 索引中的 token 进行查询，而 Term Query Parser 是针对 Solr 索引中可读版本的 Term 进行查询。

在某些域中，比如 numeric field（数字域），它们为了获取更高的查询效率内部使用了 trie 结构来存储数据，Term Query Parser 会接收可读版本的数字（比如 1.5），而 Raw Query Parser 会接收机器可读版本的 token（即索引内部实际存储的域值的表示形式），比如 Integer 域的数字 1 在 Solr 的 trie 结构中实际存储的是 `#8;#0;#0;#0;#1;`。

下面的两个查询都表示返回索引文档中 myintfield 域上包含数字 1 的：

```
{!term f=myintfield}1
{!raw f=myintfield}`#8;#0;#0;#0;#1;
```

你可能会觉得在标准的搜索应用程序中，应该很少用到 Raw Query Parser 的高级特性，对的，没错！但 Solr 允许你使用 Raw Query Parser 来满足你的特殊需求。Solr 的优雅就在于它为用户提供了高性能的查询然而并不要求用户完全理解其内部的索引存储结构。因此，你应该优先考虑使用 Term Query Parser，除非对 Solr 底层的索引存储结构很熟悉，那么可以考虑使用 Raw Quer Parser。

### (3) Function VS Function Range Query Parser

Solr 其中一个很强悍的功能就是 Function Query，Function Query Parser 就是用于构建 Function Query。Function Query 会在查询时动态的生成值，比如动态计算计算地理空间距离、进行数学计算、进行字符串转换或者在你自定义的 Function 插件中运行任意代码。使用语法如下所示：

```
{!func}log(foo) // 对 foo 执行数学中的 log 函数
```

Function Range Query Parser 用于创建 Function Range Query，使用语法如下：

```
{!frange l=1000 u=50000}myfield
```

其中：

- l: 表示 function 函数计算的最小值；
- u: 表示 function 函数计算的最大值；
- incl: 表示是否包含最小值的上边界，默认为 true；
- incu: 表示是否包含最大值的下边界，默认为 true。

下面是一个完整的使用示例：

```
fq={!frange l=0 u=2.2} sum(user_ranking,editor_ranking)
```



#### (4) Nested Query Parser

以上我们都是在学习单独的某一个 Query Parser，你可能已经知道对于指定的 Query 类型应该选择使用哪种 Query Parser，但如何结合多个 Query parser 来构建一个 Query 呢？

Nested Query Parser 提供了一个特殊的 `_query_` 操作符，它允许将其与其他 Query Parser 进行适配。`_query_` 的语法如下所示：

```
_query_:"[QUERY]"
```

其中 `[ QUERY ]` 部分表示任意你使用 `q` 参数或 `fq` 参数构建的查询。考虑如下一个查询：

```
/select?q=category:("technology" OR "business") AND
_query_: "{!edismax qf=title^10 category^4 text}solr lucene hadoop mahout"
```

在这个示例中，eDisMax Query Parser 将 “solr lucene hadoop mahout” 这个原本的 Phrase Query 进行功能增强，指定了查询域并为每个域指定了权重值，然后这整个查询又被引号所包裹，这意味着被包裹的查询中的特殊字符必须使用反斜杠进行转义。但你需要注意的是，在大部分情况下，没有必要使用 `_query_` 语法，因为 Lucene Query Parser 和 eDisMax Query Parser 内部已经能够自动根据 LocalParam 来推断是否需要 Query 转换，除非查询表达式确实是复杂难懂，你才可能会需要使用 `_query_` 语法来构造 Nested Query 来确保查询表达式是按照你期望的方式被解析的。

除了特殊的 `_query_` 操作符语法，Solr 还包含了内置的 Nested Query Parser，它支持对 Nested Query（嵌套查询）的解析。针对 Nested Query 的一个典型的本地参数就是 `query`，示例如下：

```
/select?q={!query v=$nestedQuery}
```

当你使用这种方式来定义你的 Query，那么就可以替代任何 Query。当你想要提前在 `solrconfig.xml` 中定义你查询中的某一部分时可能会很有用，比如：

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="nestedQuery">{!func}product(popularity, 0.25)
  </str>
```

在上面这个示例中，我们提前在 `solrconfig.xml` 中定义了一个 `nestedQuery`，这里其实就是提前定义了一个 Function Query，它可以单独作为一个 Query，你也可以使用 Nested Query Parser 将其替代到任意 Query 中，比如：

```
q={!query v=$nestedQuery}
```

这里通过 `v = $nestedQuery` 引用了事先定义的 Function，并与 `q` 参数构造的 TermQuery（也可能是 Phrase Query）一起构成了一个复杂的 Function Query。也就是说 `q` 参数原本构造的 TermQuery 就被隐式的转换成 Function Query。



`_query_` 操作符和 `!query` 语法属于高级用法，然而在实际中，大部分的搜索应用程序的查询表达式不会这么复杂。

Nested Query Parser 允许使用多种 Query Parser 来定义你查询中的多个部分子查询，然后通过变量引用组合到 `q` 参数构造的主查询中，这对于为你的 Function Query 进行加权会特别有用。

### (5) BoostQuery Parser

Boost Query Parser 允许为任意匹配查询条件的索引文档定义权重值，但它并不会过滤掉不匹配 Boost Query 的索引文档，只是不匹配 Boost Query 的索引文档不会添加权重。我们回顾下 `q` 参数构造的查询，它通常会根据查询表达式中指定的过滤条件过滤掉不匹配条件的索引文档，然后对匹配条件的索引文档进行相关性评分。而 Boost Query Parser 允许构造一个查询作为过滤条件，然后匹配条件的索引文档会被添加指定的权重值，而不匹配条件的索引文档不会被加权但不会被过滤掉。这就好比在企业招聘信息中经常看到的那样：有 Scala 或 Spark 工作经验者优先考虑。符合这项条件的面试者会拥有加分项，但不代表不符合这个要求的面试者就直接被过滤掉，它仅是一个加分项，我们的 Boost Query Parser 也是同样的道理。Boost Query Parser 的这个特性对于你想要实现类似“最近新上架的商品优先靠前显示”这样的需求时，会非常有用。

Boost Query Parser 的使用语法如下所示：

```
{!boost b=1000}shouldboost:true
{!boost b=log(popularity)category:trending
{!boost b=recip(ms(NOW,articedate),3.16e-11,1,1)}category:news
```

你还可以通过 Nested Query (嵌套查询) 来结合多个 Query Parser 来构建一个复杂的 Query，例如：

```
/select?q=_query_:"{!edismax qf=title content}data science" AND
_query_:"{!boost b=log(popularity))*:*" AND
_query_:"{!boost b=recip(
ms(NOW,articedate),3.16e-11,1,1)}category:news"
```

上面的示例中，我们将两个 Boost Query Parser 和一个 eDisMax Query Parser 通过 `_query_` 和 AND 操作符结合在一起。

### (6) Prifex Query Parser

Prefix Query Parser 可以用来代替通配符查询，它的语法如下所示：

```
{!prefix f=myfield}engin
```

上面这个查询等价于 Lucene 中的 `myfield: engin*`，会匹配 `myfield` 域上所有以 `engin` 开头的索引文档。需要注意的是，Prefix Query Parser 构造的 PrifexQuery (前缀查询) 会直接拿前缀字符串与索引中的 Term 进行比较查询，不会对前缀字符串进行分词处理，因此，你

需要清楚索引中的 Term 是如何表示的。但由于这种前缀查询性能不好，所以一般不建议使用。

### (7) Spatial Query Parser

Solr 中提供了丰富的地理空间方面的处理功能，它允许你在索引创建时或查询时定义一个地理坐标点或者一个地理形状（比如圆形，四边形、多边形等）。利用这个坐标点，可以过滤出距指定坐标点在指定范围内的索引文档。比如，给定北京市的经纬度，你可以过滤出方圆 5000 米之内的酒店。Solr 中包含了两种 Query Parser 来处理地理空间查询：Spatial Box Query Parser (bbox) 和 Spatial Filter Query Parser。

### (8) Join Query Parser

Join Query Parser 允许你通过跨多个 Document 集合执行一个伪连接来运行一个子查询。举个例子，你可以依据不同的 Document 集合运行一个子查询来限制一个查询以及约束你的原始查询返回的结果集必须在子查询的结果集集合中存在。这些 Join 连接功能可以跨多个 Document，而这些 Document 可以存在于不同的 Core 中，Join 和 Function Query 都属于 Solr 的高级功能，这些留到后面章节再做讲解，这里暂时略过。

### (9) Switch Query Parser

Switch Query Parser 允许你基于一些逻辑条件来选择应用哪个 Query 或者 Filter Query，这个操作类似于大部分编程语言里的 switch 语句。下面的示例演示了 Switch Query Parser 的使用语法：

```
fq={!switch
case.day='date:[NOW/DAY-1DAY TO *] '
case.week='price:[NOW/DAY-7DAYS TO *] '
case.month='date:[NOW/DAY-1MONTH TO *] '
case.year='date:[NOW/DAY-1YEAR TO *] '
case.else='*:*'
v=$withinLast}
```

上面示例定义了多个 Filter Query，实际需要根据变量 \$withinLast 的值来决定使用哪个 Filter Query，你可以在 q 参数的主查询中指定 withinLast = day，那么等价于在主查询中添加了 date:[NOW/DAY-1DAY TO \*] 这个 Filter Query。如果给定的参数值都不配置，那么就以 case.else 的为准。当你想要根据自己的业务参数执行不同的 Filter Query 时，Switch Query Parser 可能会比较有用。

### (10) Surround Query Parser

Surround Query Parser 用于构建一个 Span Query，Span Query 会依赖于每个 Term 之间的位置关系，它使得你的查询不再依赖于 Term 的精确匹配，它允许你对 Term 跨多个 Term 进行移动，移动方向可以向前、向后，在规定的移动次数内达到用户输入的字符串效果就判定为匹配。Surround Query Parser 使用两个特殊的操作符 w（无顺序）和 n（有顺序）来表示在匹配 Term 的时候是否考虑 Term 的位置顺序，即给定的查询字符串是 “a b”，如果索引

中包含的 Term 经过指定次数的移动后，能与用户输入的查询字符串保持一致，那么就判定该索引文档匹配查询条件应该被返回，这时候就涉及 Term 位置顺序问题。可能经过  $N$  次移动后只能得到“b a”。如果使用的是  $w$  操作符，那么就判定为匹配；如果使用的是  $n$  操作符，那么就判定为不匹配，即必须跟用户输入的查询字符串中包含的 Term 的出现顺序也保持一致。

考虑以下示例：

```
{!surround}3w(solr, action)
{!surround}5n(solr, action)
{!surround}solr 3w action
{!surround}solr 3n in 2w action
```

上面第一个示例表示查找索引文档中包含 solr 和 action 这两个 Term，并且 solr 和 action 这两个 Term 之间经过最多 3 次移动后能匹配到“solr action”，由于使用的是  $w$  操作符，即表示经过最多 3 次移动之后最后匹配到“action solr”也是符合要求的。第 3 个示例与第 1 个示例是等价的，只是写法不同。第 2 种使用了  $n$  操作符，即表示最后匹配到的必须是 solr 在前 action 在后，且最多移动 5 次。第 4 个示例演示了在多个 Term 之间指定间隔的其他 Term 的最大个数。

此外，Surround Query Parser 还支持使用 AND OR 操作符，比如：

```
{!surround}3w(solr AND action)
{!surround}5n(solr AND in OR action)
```

Surround Query Parser 存在一个限制，就是它缺少对用户输入的查询文本进行分词处理。很遗憾，它跟 Term Query Parser 类似，都是直接根据输入的 Term 与 Solr 索引中的 token 进行比较查询的，即便该域类型上定义了分词器也不会进行分词。尽管 Surround Query Parser 也支持通配符（`hell*w?rld`）和设置权重（`fieldName^10`），但由于它不支持对查询文本进行分词，使得在关键字搜索场景下，这两项功能变得没什么用处。

### （11）Max Score Query Parser

当你使用 Lucene Query Parser 对一个查询进行评分时，每个 Term 的得分都会被纳入最终得分的考虑因素中，并且最终会求所有匹配 Term 的文档的综合得分作为文档的最终得分。有时候，这种评分方式会显得更公平，但某些情况下，可能取其中多个匹配条件命中的文档的最高分作为最终得分，相比于 Lucene Query Parser 取总分而言，会显得更有用。Max Score Query Parser 的使用语法如下所示：

```
{!maxscore}term1 term2 term3
```

在上面这个查询中，我们输入了 term1、term2、term3 这 3 个搜索关键字，然后我们通过“!maxscore”操作符告诉 MaxScoreQParserPlugin，在计算匹配文档的最终得分时采用求最大值的方式。你也可以使用 `_query_` 操作符将 Max Score Query Parser 与其他 Query Parser

结合起来进行综合运用，比如：

```
/select?q=one OR two OR _query_: "{!maxscore v=$maxQ}"&
maxQ=three OR four OR five
```

在上面这个查询中，maxQ 查询会按照求最大值方式计算文档评分，而我们的 q 参数构造的主查询仍然以求总和的方式计算文档评分。

(12) Collapsing Query Parser

Collapsing Query Parser 提供了移除查询返回的结果集中的重复 Document 的能力，前提是索引文档中你指定的某些域拥有唯一标识的值。这种能力被称为 Field Collapsing，当你想要确保查询返回的结果集中的文档不会出现重复时，会非常有用。实现 Field Collapsing 有两种方式，第一种就是使用一种被称为 result grouping 的功能套件，它允许你根据指定域对查询返回的结果集进行分组，并且每组值返回指定数量的索引文档。另外一种方式就是利用 Collapsing Query Parser 实现。关于 Group 和 Field Collapsing 会在后续章节进行讲解。

(13) Complex Phrase Query Parser

顾名思义，这是一个用来构建 Phrase Query（短语查询）的查询解析器，内部是使用 Lucene 的 ComplexPhraseQueryParser 类实现的。但 Complex Phrase Query Parser 可不仅能够用来构建 Phrase Query 它还支持用来构建 Wildcard Query（通配符查询）、Span Query（跨度查询）。它提供了如表 5-10 所示可选的设置参数。

表 5-10 可选参数

参 数	描 述
inOrder	设置为 true 则表示强制要求 Phrase Query 严格按照 Term 定义的顺序去匹配，默认值为 true
df	df 表示默认查询域，默认值为 text

使用示例：

```
// 通配符查询
{!complexphrase inOrder=true}name:"Jo* Smith"
// 通配符查询与短语查询的组合
{!complexphrase inOrder=false}name:"(john jon jonathan~) peters*"
// 短语查询加特殊字符转义
+_query_:"{!complexphrase inOrder=true}manu:\"a* c*\" +_query_:{!complexphrase
inOrder=false df=name}\"bla* pla*\""
```

使用通配符查询时通配符表达式匹配到的 Term 的个数会严重影响查询的性能，比如说查询“a\*”会匹配所有以 a 开头的 Term，这会生成大量的 OR 查询语句。所以查询时请谨慎使用通配符。通配符查询中的前缀字符串长度越小匹配到的文档数目会越多，但查询性能会越差，返回的查询结果集的相关性也越低。

由于通配符查询可能会生成大量的 OR 语句，因此可能会提示 Boolean Clauses 已超出限制，此时你需要在你的 `solrconfig.xml` 文件中酌情调大此参数值：

```
<maxBooleanClauses>4096</maxBooleanClauses>
```

建议不要在此 Query Parser 上使用停用词过滤器，假如我们添加了停用词，且停用词字典里包含了 `the`、`up`、`to` 这些停用词，而你索引的文档的 `features` 域的某个域值为 “Stores up to 15 000 songs, 2500 photos, or 150 yours of video”，那么当你执行下面这个查询时会正常返回结果：

```
q=features:"Stores up to 15,000"
```

但是当你执行下面这个查询时，却匹配不到任何索引文档：

```
q=features:"sto* up to 15*"&defType=complexphrase
```

因为 `SpanNearQuery` 查询对停用词处理支持的并不好。如果某些使用场景下要求你必须使用停用词，那么此时可以通过自定义的 `Filter` 解决此类问题。

至此，你已经了解了 Solr 中内置的众多 Query Parser 的用法，除了使用这些内置的 Query Parser，你还可以自己定义 Query Parser 插件，虽然对于大部分用户来说，使用内置的这些 Query Parser 就能满足他们的需求，但如果确实有自定义 Query Parser 的需求，你可以通过继承 `QParser` 抽象类自定义你的 Query Parser，并继承 `QParserPlugin` 抽象类，自定义你的 `QParserPlugin` 插件，每个 `QParser` 通常会对应一个 `QParserPlugin` 类。`QParserPlugin` 插件类编写完成后，还需要在你的 `solrconfig.xml` 配置文件中对其进行注册，示例如下：

```
<queryParser name="myparser" class="com.mycompany.MyQParserPlugin"/>
```

其中 `name` 属性用于指定 Query Parser 的名称，`class` 属性用于指定 Query Parser 实现类的完整访问包路径，前提是需要将自定义的 `QueryParserPlugin` 类打包成 jar 并放置到当前 `core` 的 `lib` 目录下，否则会提示插件类找不到导致插件类加载失败，最终会导致 `Core` 启动失败。

## 5.9 Query VS Filter Query

学习了 Solr 内置的各种 Query Parser 之后，你还需要好好理解 Query 和 Filter Query 是如何工作的。比如 Query 和 Filter Query 有何区别？他们直接是如何相互影响的？他们是如何决定查询的性能已经最终返回结果集的相关性的？Solr 查询由两个重要的查询操作组成，他们匹配用户查询请求参数以及对匹配的结果集进行排序，以便于匹配度较高的前几个索引文档会被返回。默认索引文档会基于相关性评分进行排序，这意味着在查询结果集被查询并收集到之后，需要一个额外的操作来计算每个匹配的索引文档的相关性评分。



### 5.9.1 fq VS q

为了高效查询匹配的索引文档以及对匹配索引文档进行评分，Solr 使用两个参数：fq 和 q。fq 参数代表 Filter Query；q 参数代表 Query。第一眼看到这两个参数感觉很难区分，因为他们拥有着相同的查询语法，并且会返回相同的查询结果集数量，正因为如此，大部分使用者觉得两者的功能是一样的，从而忽略了 fq 的存在。通过了解并理解 fq 和 q 之间的区别将有助于你更高效地执行查询。

q 和 fq 之间到底有着什么样的区别呢？fq 只有一项单一的职责：对匹配的索引文档进行过滤限制，不会对索引文档进行相关性评分操作。而 q 参数拥有两项职责：

- 根据用户传入的查询条件匹配符合条件的索引文档；
- 使用相关性算法根据 Term 列表对匹配到的索引文档进行相关性评分，这些 Term 列表可能是用户传入的，也有可能是对用户输入的查询文本字符串经过分词器处理后得到的。

因此，q 参数作为一个特殊的过滤，它会告诉 Solr 在计算相关性评分时什么 Term 应该考虑进去。正因为这种差别，人们更倾向于将输入的查询关键字传递给 q 参数，然后通过 fq 参数自动生成 Filter Query。

从 q 参数构造的主查询中分离出来的 Filter Query，会经常在查询之间被重用，因为 Filter Query 会在 Filter 缓存区缓存 Filter Query 匹配到的索引文档。由于 q 参数构造的主查询需要对匹配到的每个索引文档进行相关性评分，而将查询某些部分分离到 Filter Query 中，那么这些被分离到 Filter Query 的部分匹配到的索引文档将不会进行相关性评分操作，这将大量减少了索引文档的相关性评分的计算工作量。

关于 Solr 查询，你需要注意的最后一点就是你可以在你的查询请求中添加多个 fq 参数，从而构造多个 Filter Query，但是 q 参数只能指定一个。考虑下面的两个查询：

```
q=keywords:solr&fq=category:technology&fq=year:2013
q=keywords:solr&fq=category:technology AND year:2013
```

上面两个查询返回的结果集是相同的，但他们的查询执行效率是不同的，因为 Filter Query 不会执行额外的索引文档的相关性评分，以及内部会为每个 Filter Query 分配独立的缓存区来缓存 Filter Query 匹配到的索引文档，那么第二次以及后续再次执行同样的 Filter Query 时，将会直接命中缓冲区中的索引文档，因为此时是在内存中查找，查询效率自然就提升了。鉴于 Filter Query 内部的这种机制，应该将那些能够过滤掉大部分无效索引文档的查询条件通过 fq 参数实现，你也可以将那些用户使用频率比较高的查询条件使用 fq 参数来实现，充分利用缓存来提升查询效率。

因为 Query 和 Filter Query 都可以用来查找索引文档，那么可能有人就要问了：Query 和 Filter Query 的执行顺序是怎样的？有人说 Filter Query 先执行，有人说 Query 先执行，也有人说 Filter Query 和 Query 是并行执行，到底是哪种？这看起来是个超级复杂的问题。



问题的答案取决于你的使用场景。

从纯技术的角度来讲，它们的执行顺序是这样的：

- ❑ 首先每个 fq 参数会在 Filter 缓存区查找索引文档，如果在缓存中存在，那么会返回命中中的 DocSet；
- ❑ 如果 fq 参数在 Filter 缓存区没有命中且 Filter 缓存开启了，那么会根据 fq 参数构造 Filter Query 从 Solr 索引中加载获取每个索引文档的 DocSet，并存入 Filter 缓存；
- ❑ 根据 q 参数构造主查询从 Solr 索引中加载匹配的所有索引文档，得到一个索引文档的集合，如果主查询返回的索引文档的内部 ID 在 Filter Query 缓存的 DocSet 中也存在，那么该索引文档就判定为应该返回给用户，然后就对该匹配的索引文档计算相关性评分；
- ❑ 如果主查询还包含其他 POST Filter（它会在 Query 和 Filter Query 执行完毕之后才执行），他们还会执行一部分的索引文档收集工作；

鉴于上面的执行流程解释，从技术层面上来讲，在 Filter 缓存开启的情况下，Filter Query 应该是先于 Query 执行的，随后 Query 和 Filter Query 在索引文档收集阶段是并行执行的，当两者的索引文档收集工作执行完毕之后，POST Filter 开始执行。

这看起来似乎很复杂，然而它的确是蛮复杂的。Solr 很好地为你隐藏了内部的这些复杂性，但是理解内部这些执行过程有助于你去优化那些开销比较大的查询。Solr 允许你非常细腻的控制哪些 Filter Query 应该被缓存以及它们的执行顺序。

## 5.9.2 Filter Query 缓存

Filter Query 通过 Filter 缓存和忽略索引文档的相关性评分工作来节省大量的查询处理时间，但并不是所有的 Filter 都是这样的。比如，你想要根据经度和纬度去过滤指定半径的地理范围内匹配的索引文档时，Filter Query 需要非常昂贵的数学计算开销。除此之外，如果你为上百万的地理位置生成不同的 Filter Query，这些上百万个 Filter Query 会对应有上百万个独立的 Filter 缓存区，那么你的内存使用率会降低。那样也可以只生成一个 Filter Query，那么对应的只有一个 Filter 缓存区，此时 Filter 缓存区就会超负荷工作，同时还可能会导致 IndexSearcher 实例自动预热过程变得非常昂贵。为此，Solr 允许你控制哪些 Filter Query 应该被缓存，以及每个 Filter Query、Query、POST Filter 之间的执行顺序。

在有些情况下，可能会有大量的 Filter Query，但它们并不值得被缓存。因为你可能限制了需要缓存的 Filter Query 的数量。当最常用的 Filter Query 在任意时刻都保持缓存命中的话，那么你的 Solr 查询性能是最优的。

为了避免 Filter 缓存被一些不太重要的 Filter Query 过度负载了，你可以显式关闭任意 Filter Query 的缓存，具体语法如下所示：

```
fq={!cache=false}id:123&
fq={!frange l=90 u=100 cache=false}
```

```
scale(query({!v="content:(solr OR lucene)"}),0,100)
```

第一个 fq 关闭了对 id: 123 查询的缓存功能, 第二个 fq 指定过滤出 (solr ORlucene) 这个查询中前 10% 比较相关的索引文档, 并关闭了对该查询的缓存功能。query 函数会执行文档查询并正常计算匹配的索引文档的相关性评分, 然后 scale 函数会对其匹配的每个索引文档的相关性评分进行缩放, 缩放后的评分取值范围在 [1, 100] 区间内, 然后 frange 操作符通过 l 和 u 两个参数限制每个匹配的索引文档的缩放后的评分范围, 从而过滤出前 10 个相关性比较高的索引文档。默认 Filter Query 的 Filter 缓存是开启的, 即 cache 参数的默认值是 true。

### 5.9.3 Filter Query 执行顺序

如果你的查询请求中包含了多个 Filter Query, 每个 Filter Query 的执行顺序会直接影响你的查询性能, 如果一个 Filter Query 提前执行并过滤掉一部分结果集缩小了查询匹配范围, 那么接下来执行的 Filter Query 会基于更少的 Document 进行二次查询匹配, 其查询速度自然就更快。相反的, 如果一个 Filter Query 必须要执行非常复杂的计算, 比如 geospatial Filter 需要过滤指定半径的范围内的坐标点, 那么此时应该给予更少的索引文档去执行昂贵的 CPU 计算, 那么就意味着那些比较昂贵的 Filter Query 需要尽量靠后执行。

当你清楚地知道 Filter Query 中哪个执行开销是最昂贵的, Solr 允许你通过指定一个 cost 参数来强制指定 Filter Query 的执行顺序, cost 参数表示对该 Filter Query 执行开销的一个估算数值, 数值越大说明该 Filter Query 的执行开销越大, 即意味着它应该越靠后执行。cost 参数的具体使用语法如下所示:

```
fq={!cost=1}category:technology&
fq={!cost=2}date:[NOW/DAY-1YEAR TO *]&
fq={!geofilt pt=37.773,-122.419 sfield=location d=50 cost=3}&
fq={!frange l=90 u=100 cache=false cost=100}
scale(query({!v="content:(solr OR lucene)"}),0,100)
```

cost 参数值没有强制要求必须是连续的数字, 默认会按照彼此之间的相对顺序来决定 Filter Query 的执行顺序。当 cost 参数值  $\geq 100$  时, 此时该 Filter Query 在 Solr 中被称为 Post Filter。

### 5.9.4 Post Filter

有时候, 某些 Filter Query 的执行开销会很昂贵, 你期望它等到 Query 和所有的 Filter Query 全部执行完毕之后再执行它。为此, Solr 中实现了一种特殊的 Filter: Post Filter。当 Query 和 Filter Query 并行收集索引文档时, 每个索引文档被 Query 或 Filter Query 所收集, 只有当每个索引文档都被收集后, Post Filter 才会随后被执行。这个特性允许执行开销比较低的 Filter Query 先执行去限制总的匹配文档数目。随后执行的开销比较昂贵的 Post Filter

会基于一个比较小的索引文档集合进行过滤查询，这在一定程度上也能提升执行开销比较昂贵的 Post Filter 的执行性能。cost 参数除了能够对 Filter Query 指定之外，其实它能够用于 Post Filter，当你将一个 Filter Query 的 cost 参数值设置的大于等于 100，那么默认该 Filter Query 就会被转换成 Post Filter。但前提是该 Filter Query 实现了 PostFilter 接口且  $\text{cost} \geq 100$ 。

你没有必要为所有的 Query 和 Filter Query 都实现 PostFilter 接口，除非确实需要将该 Filter Query 放置后最后执行。如果想要自定义 Post Filter，那么你需要实现 PostFilter 接口。

## 5.10 Solr 返回结果

通过对前面章节的学习你已经知道如何使用 Query Parser 来构造 Query。通过 Query 来查询返回你期望的索引文档，通常可能还需要指定每个文档应该返回哪些域供客户端页面显示，基于文档的相关性评分对索引文档进行排序，以及对查询返回的结果集进行分页，甚至你可能还需要设置返回的响应结果集的输出格式。下面的将会对这些内容进行讲解。

### 5.10.1 设置响应输出格式

默认 Solr 查询返回的响应数据输出格式为 XML。对于 Solr Server 端来讲，采用哪种输出格式都是无关紧要的，至于客户端期望返回什么数据格式，则需要通过指定 wt 参数来强制要求 Solr Server 返回指定的数据格式。Solr 默认支持的响应输出格式有 XML、JSON、Ruby、Python、Binary Java、PHP、CSV 等，甚至可以是自定义的响应输出格式。表 5-11 列出了 Solr 中默认支持的响应输出格式。

表 5-11 Solr 默认支持的响应输出格式

Write Type	实现类	Write Type	实现类
csv	CSVResponseWriter	phps	PHPSerializedResponseWriter
json	JSONResponseWriter	python	PythonResponseWriter
xml	XMLResponseWriter	ruby	RubyResponseWriter
xslt	XSLTResponseWriter	javabin	BinaryResponseWriter
php	PHPResponseWriter		

下面是一个设置 wt 参数的演示示例：

```
/select?wt=json&indent=true&q=*&fl=id,title&rows=2
```

响应输出内容如下：

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 1,
    "params": {...},
```

```

"response":{"numFound":1000,"start":0,"docs":[
  {
    "id":"1",
    "title":"solr in action",
  },{
    "id":"2",
    "title":"lucene in action",
  }
]}
}}

```

正如上面示例演示的那样，Solr 中的响应输出格式设置非常简单，你只需要设置 `wt` 参数值即可，如果需要自定义响应输出格式，那么你需要编写自己的 `Response Writer` 实现类。为此，需要创建一个类实现 `QueryResponseWriter` 接口，并重写其 `init`、`getContentType`、`write` 这 3 个函数，然后，你必须在 `solrconfig.xml` 中注册刚刚创建的那个 `Response Writer` 类，配置示例如下所示：

```

<queryResponseWriter name="myResponseWriter"
  class="xxx.xxxx.MyResponseWriter" />

```

对于大多数编程语言来说，都提供了非常方便使用的类库用于处理 JSON 和 XML 的格式化，因此完全没有必要去浪费时间实现自己的 `Response Writer` 重新造轮子，除非自带的 `Response Writer` 确实不能满足你的需求时，你才需要考虑自定义。当然也有可能，你的项目中有很强安全性要求，比如可能想要实现一个 `Response Writer`，它在将响应结果集返回给客户端之前需要先进行加密处理。

## 5.10.2 选择返回域

当你在 `schema.xml` 中定义域时，你可以指定一个可选的属性 `stored`，表示是否是一个存储域。在对存储域建立索引的同时还会将域的原始值写入到索引中，虽然这样会额外占用一些硬盘空间，但带来的好处是存储域的值可以随响应结果集一并返回给用户用于展示。但是假如该域不是存储域，那么通过 `fl` 参数返回给用户并没有什么意义，因为只有存储域返回才有意义。

### 1. 返回存储域

当在 `schema.xml` 中定义域时，你可以指定一个可选的属性 `stored`，表示是否是一个存储域。在对存储域建立索引的同时还会将域的原始值写入索引中，虽然这样会额外占用一些硬盘空间，但带来的好处是存储域的值可以随响应结果集一并返回给用户用于展示。但是假如该域不是存储域，那么通过 `fl` 参数返回给用户并没有什么意义，因为只有存储域返回才有意义。

`fl` 参数可以指定多个你期望返回的域，多个域名称之间使用逗号分隔，比如：

```

/select?...&fl=id,name

```

```
/select?...&fl=*
```

上面第一个示例表示返回 id 和 name 这两个域，第二个示例表示返回文档中的所有 stored = true 的域，除了返回存储域之外，Solr 还允许你将动态生成的值作为索引文档的一个伪域来返回给用户。

## 2. 返回伪域

除了返回存储域之外，每个索引文档中比较有价值的信息其中的一个就是文档的评分，你可以在 fl 参数中指定 score 这个暗含的“伪域”，比如：

```
/select?...&fl=*,score
```

上面示例表示返回索引文档中的所有存储域，外加每个文档的评分，这个评分是根据每个域的域值相关性动态计算而来。文档的评分并不会自动返回给用户，你需要显式指定 score 这个“伪域”才会返回每个索引文档的评分。

当然，你还可以对指定域执行一个 Function（函数），将函数动态计算得到的值作为一个伪域返回给用户，比如：

```
/select?...&fl=id,sum(integerField,10)
```

上面这个示例中，将 integerField 这个域的域值加上 10 然后作为一个伪域随着 id 域一并返回给用户，这里用到额 sum 求和函数其实就是执行了一个 Function Query，有关 Function Query 的相关知识，后续章节再做讲解。

## 3. 文档转换器

有时候，在查询匹配的索引文档写入到 Response 中之前，往索引文档中添加一些额外信息可能会比较有用，比如你可以添加查询计划信息、在分布式搜索中文档属于哪个 shard 以及索引文档的内部 ID。这类信息在 Solr 内部是通过 Document Transformer（文档转换器）转换而来的，Document Transformer 可以按照如下方式被调用：

```
/select?...&fl=*,[explain],[shard]
```

如表 5-12 所示，列出了 Solr 中支持的通用的 Document Transformer。

表 5-12 通用的 Document Transformer

示 例	含 义
[docid]	Lucene 内部索引文档的 ID (Integer 类型)
[shard]	匹配到的索引文档所属 shard 的 ID
[explain]	查询执行计划相关信息，详细描述打分过程
[explain style = nl text html]	查询执行计划相关信息，style 属性用于指定输出格式，支持 3 种格式：nl、text、html
[value v=? t = int double float date]	一个指定的值，该值每个索引文档都相同

explain 的 style 有 3 种可选值, 其中 nl 格式表示分多行显示; text 和 html 表示单行显示; html 风格会添加额外的 HTML 标签元素; text 只会显示纯文本字符串。

当你想要通过低级别的 API 与 Lucene 的索引文档进行交互时, 可能会用到 [docid], 对于大部分 Solr 用户来说, 返回 [docid] 并不常用。在分布式搜索中, 当你想要查找匹配到的索引文档在哪个分片上、该分片在哪台 Solr Server 上以及每个索引文档存在于哪个 core 中, 这个时候指定 [shard] 会比较有用。添加 [explain] 这个伪域有助于你理解匹配到的每个索引文档的得分是计算得来的, 同时它也能帮助你分析你的查询中可能存在的问题, 有点类似 MySQL 中的 SQL 执行计划, 其实 MapReduce 中也有类似的执行计划概念。[value] 这个伪域用于为匹配的每个索引文档指定一个固定的静态值。

除了可以使用这些 Solr 内置的 Document Transformer 之外, 你还可以自定义你自己的 Document Transformer 插件, 此时你需要继承抽象类 org.apache.solr.response.transform.DocTransformer 并实现它的 transform 函数。为了使你自定义的 Document Transformer 能够在 Solr 中可用, 你还需要一个 Document Transformer 的 Factory 工厂类 (需要继承 org.apache.solr.response.transform.TransformerFactory 类), 最后你需要在 solrconfig.xml 配置文件中注册你自定义的 Document Transformer 插件, 配置示例如下所示:

```
<transformer name="magic" class="magicTransformer" >
<string name="yourSetting">xxxxxx</string>
</transformer>
```

虽然 Document Transformer 并不是 Solr 中比较常用的功能, 但 Solr 依然提供了扩展点允许用户在文档返回给 Response 之前操作 Document, 比如添加删除 Document, 编辑 Document 的域等。

#### 4. 返回域别名

除了可以将动态生成的值作为伪域返回之外, Solr 还提供一种在你返回搜索结果集之前对域定义别名的能力。这有点类似于 SQL 里的 “select tid as id, stu\_name as sname” 为返回表字段定义别名。定义别名语法如下所示:

```
/select?...&fl=id,betterFieldName:actualFieldName
```

betterFieldName 表示定义的别名, actualFieldName 表示 schema.xml 中实际存在的域的名称, 当你在使用动态域, 比如有一个名称为 fieldname\_t\_is 的动态域, 当你在返回该域之前希望返回一个更友好的名称给用户, 或者你不想将域的真实名称返回给用户时, 这个特性会比较有用。通常没有必要返回索引文档上的所有存储域, 除非你确实有需要, 因为一次查询中返回太多域会增加额外的查询开销。

### 5.10.3 分页查询

一个查询可能会匹配到很多索引文档 (几百甚至几百万), 但通常只会返回其中的一页



给用户。Solr 就是为了实现非常高效的（毫秒级的）在数百万甚至数十亿的索引文档中进行查询而设计的，但 Solr 并没有对返回大量索引文档做优化，Solr 比较擅长于返回几十条或几百条索引文档，但是当你返回几千条索引文档，查询的请求时间会明显加大、吞吐量会明显放缓，使用 Solr 的最佳实践就是每次只返回结果集中的前几条索引文档作为一页数据返回给用户，通过这种方式，Solr 的每次查询请求只返回限定数据的索引文档，因此，这样就避免了一次查询返回大量数据占用了大量系统资源，并且拖缓其他查询的可能性。在 Solr 中实现查询分页需要用到两个请求参数：`start` 和 `rows`。`start` 参数表示查询结果集返回的偏移量，它是从零开始计算的。`rows` 参数表示一页查询返回多少个索引文档给用户。下面的示例演示了 `start` 和 `rows` 参数的使用：

```
/select?q=*&sort=id&fl=id&rows=5&start=5
```

可能返回的结果如下所示：

```
{...
  "response":{
    "numFound":100000,
    "start":5,
    "docs":[
      {
        "id":"6"},
      {
        "id":"7"},
      {
        "id":"8"},
      {
        "id":"9"},
      {
        "id":"10"}
    ]
  }
}
```

这里的 `start` 和 `rows` 参数类似于 MySQL 里的 `limit` 语法里涉及的两个参数，两者表达式的含义差不多，你可以类比 MySQL 里的分页语法参数去理解。但需要注意的是，`start` 参数不应该大于匹配到的索引文档的总数，否则分页查询将得不到数据。

## 5.11 Solr 排序

当一个查询执行完成之后，返回的查询结果集会按顺序排列，Solr 中默认是按照文档的相关性评分从高到低进行排序的，但是你其实可以基于任何内容进行排序，比如日期、词、数字、Function（函数），本节将会讲解 Solr 中的查询结果集排序机制。

### 5.11.1 根据域进行排序

当你运行一个关键字查询时，返回的查询结果集默认会按照相关性评分降序排列，即得分高的会排在前面。如果两个索引文档的得分相同，那么此时会再按照索引文档的内部

ID 进行升序排列。如果你没有设置对索引文档进行打分，那么可能索引文档没有相关性评分，这意味着所有索引文档的得分都是相同的，最终索引文档就会默认按照内部的 DocID 进行升序排序。默认的这种排序行为可以很轻易地通过在查询请求中添加 `sort` 参数来覆盖：

```
sort=someField desc, someOtherField asc
sort=score desc, date desc
sort=date desc, popularity desc, score desc
```

上面第一个示例表示首先按照 `someField` 降序排序，再按照 `someOtherField` 域升序排序；第二个示例中会首先按照索引文档的相关性评分降序排序，然后再按照 `date` 域降序排序；第三个示例中会首先根据 `date` 域降序排序，再按照 `popularity` 域降序排序，而后再按照索引文档的相关性评分进行排序。根据指定域排序的语法很简单，基本原则就是：排序的规则有 `asc`（升序）和 `desc`（降序），多个排序域用逗号分隔。如果你想要显式指定按照 DocID 进行排序，那么你可以这样指定：`sort = _docid asc`。

需要注意的是，任何你想要按照它排序的域必须是在 `schema.xml` 中定义了 `indexd = true`。你应该知道 Solr 排序是基于索引中 Term 的顺序来排列的，比如你有一个 `string` 类型的域，索引的值有 1、2、3、10、20、30，然后你根据它进行升序排序，那么你得到的顺序将是 1、10、2、20、3、30，而不是 1、2、3、10、20，因为对于字符串来说默认比较大小是按照字符的 ASCII 值的大小来进行比较的。此外，由于有些语言的默认排序规则与其他语言不同，Solr 内置提供了一个 `TokenFilter` 用于指定语言的字符排序规则，详细信息请查看 <http://wiki.apache.org/solr/UnicodeCollation>。如果你的域使用了该 Token Filter，那么你就可以对当前特殊的语言进行正确排序。

### 5.11.2 缺失值处理

有一个比较极端的情况可能会比较重要，那就是当某个域的域值缺失了该如何对其排序。由于 Solr 默认并不强制要求每个域的域值不存在，除非你在 `schema.xml` 中对 `<field>` 元素配置了 `required = true`。当你对于一个域值缺失的域进行排序时，可能期望它排在最前面后者最后面，由于用户的需求是千变万化的，所以 Solr 允许你选择其中任何一种适合的情况。而你只需要在 `schema.xml` 中域定义上添加 `sortMissingLast` 和 `sortMissingFirst` 属性，示例如下所示：

```
<fieldType name="string" class="solr.StrField"
sortMissingLast="true"
sortMissingFirst="false" />
```

如果 `sortMissingLast` 设置为 `true`，那么所有该域的域值缺失的索引文档将会排在最后面而不管排序方向是 `asc` 还是 `desc`。相反，如果 `sortMissingFirst` 设置为 `true`，那么所有该域的域值缺失的索引文档将会排在最前面。默认情况下，这两个属性的默认值都是 `false`。在默认设置下，`asc` 升序排序时，域值缺失的索引文档会排在最前面；`desc` 降序排序时，域值缺

失的索引文档会排在最后面。

### 5.11.3 排序的内存占用

关于 Solr 排序需要说到的最后一点就是：Solr 排序是一个内存密集的处理。为了实现索引文档进行排序，Solr 使用了 Lucene 的域缓存，当排序操作第一次被执行时，Lucene 的域缓存会加载域内部的唯一值并放置到内存中，这意味着基于指定域进行排序需要消耗大量内存，这也意味着对指定域排序第一次执行时需要构建内部的缓存区结构，这可能会导致第一次执行排序时会比较缓慢，但这不是说你不能对索引文档进行排序，而是你需要关心的是是否有足够的内存来执行排序。此外，你可能想要提前预热 Solr 实例以便能提前预热你的缓存，这部分内容将会在后续章节介绍。

## 5.12 调试查询结果

尽管你正走在成为一个有经验的 Solr 使用者的路上，但当看到查询结果时，你有时仍然会感觉很迷惑。比如有时候查询返回的结果集数目跟你预期的不一样，可能是 Query Parser 使用不当，有时候会感觉索引文档评分与你预期的不一样，有时候可能会感觉查询耗时很长。值得庆幸的是，Solr 提供了一个特殊的查询组件：DebugComponent（调试组件），通过它，你可以在遇到上述情况时得到帮助。

### 5.12.1 返回调试信息

了解 Solr 查询内部执行原理最简单的方式就是通过传递 `debug = true` 请求参数来启用 Debug 调试模式，Debug 参数设置为 true 会激活查询请求的 DebugComponent 调试组件，该调试组件返回一些调试信息。使用示例如下所示：

```
/select?q=*&rows=3&debug=true
```

返回的调试信息如下：

```
{
  "responseHeader": {...},
  "response": {... docs": [
    {
      "id": "1"},
    {
      "id": "2"},
    {
      "id": "3"}
  ]},
  "debug": {
    "rawquerystring": ".*.*",
    "querystring": ".*.*",
```

```

    "parsedquery": "MatchAllDocsQuery(*:*)",
    "parsedquery_toString": "*:*",
    "explain": { ... },
    "QParser": "LuceneQParser",
    "timing": { ... }
}

```

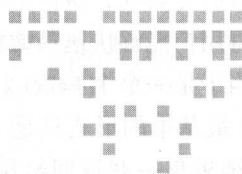
`querystring` 表示用户传入的查询表达式, `rawquerystring` 表示用户传入的原始查询文本字符串, `parsedquery` 表示用户传入的查询文本经过解析后的内部表示形式, `parsedquery_toString` 即 `parsedquery` 调用 `toString` 后的结果, 主要是为了方便用户阅读。 `Explain` 中包含的当前查询的执行计划信息, `QParser` 表示当前使用的是哪个查询解析器, `timing` 中包含的是查询的各个阶段的耗时情况信息。

### 5.12.2 开启调试模式

除此之外, 你还可以通过 `debug` 参数的其他选项来控制只返回调试信息中的某一部分信息, 除了可以设置 `debug = true`, 还可以设置 `debug = results`, 表示只返回关于查询文本被查询解析器解析相关的那部分信息。如果你设置 `debug = results`, 那么会看到关于索引文档的相关性评分的详细解释信息。如果你设置 `debug = timing`, 表示只返回查询的各个执行阶段使用的查询组件的执行耗时信息。调试查阅索引文档的相关性评分的解释信息有助于你分析查询执行过程并适当作出调整进行优化。当查询耗时很长或者查询返回的结果集相关性不太高时, 你可能需要打开 Solr 查询的调试模式, 它可以帮助你分析查询, 从而构建更高效的查询。

## 5.13 本章总结

在本章中, 我们学习了 Solr 查询的基本概念和 Solr 相关性的相关理论知识, 然后学习了 Solr 中内置的各种 Query Parser 查询解析器, 通过这些强大的查询解析器有助于你构建自己的查询。接着我们学习了在 Solr 中如何提交一个查询请求以及如何返回查询结果, 如何返回指定的域, 如何进行分页查询以及如何对查询结果集进行排序。最后我们了解了如何开启查询调试模式使用 `DebugComponent` 查询组件来调试你的有问题的查询。



## 第 6 章 *Chapter 6*

# Solr Facet

通过第 6 章，你将可以学习到如下内容：

- ❑ 理解什么是 Facet;
- ❑ Facet 简单示例搭建;
- ❑ 学习使用 Field Facet;
- ❑ 学习使用 Query Facet;
- ❑ 学习使用 Range Facet;
- ❑ 学习使用 Facet Filter;
- ❑ 学习使用 key/tag/ex 参数实现 Multiselect Facet。

## 6.1 理解 Facet

Faceting 是 Solr 中比较强大的一个功能，尤其是将其与传统的关系型数据库或者 NoSQL 数据库相比，你会越发觉得它的强大。Faceted Search 又称 Faceted Navigation (Faceted 导航) 或 Faceted Browsing (Faceted 浏览)。它允许你运行一个查询，而后基于索引文档的某个维度或方面对查询匹配的索引文档进行高标准的分类。它允许你选择一个 Filter 对查询结果集进行过滤。

当你在一个新闻网站搜索时，可能希望能够通过时间帧帮你过滤出部分结果，比如过去 1 小时、过去 24 小时、上一周。或者通过分类帮你过滤，比如政治、科技和经济。当你在一个招聘网站上搜索，可能希望能够根据城市、工作类型、行业、公司名称等来过滤出部分你感兴趣的工作职位，并且能够显示每个分类下匹配的结果总数量。因为我们只能在网站

上展示有限个数的分类项，所以大部分搜索程序经常会将那些比较热门流行的分类放在前面显示，从而使得用户能够快速鸟瞰整个搜索结果集，而不用单击每个分类进去查看。这里的每个分类就相当于是一个个 Facet（维度）。

尽管 Facet 最基本的形式只是一个域的值列表，但 Solr 中的 Facet 允许你将一些动态元数据随着查询结果集一并返回给用户。同时 Solr 还提供了很多 Facet 高级功能，比如基于 Function 计算结果值的 Facet、基于区间值的 Facet、基于任意查询的 Facet。Solr 还支持分层多维度的 Facet 以及多选的 Facet，即便该 Document 已经被 Filter Query 过滤掉，依然能够被统计。

如果上面的概念比较官方不够直白，不好理解的话，那么我举例说明吧。比如学生可以按班级来分类、可以按性别来分类、可以身高来分类、可以按年龄来分类、可以按考试分数来分类、可以按兴趣爱好分类、可以按出生地址分类等，上面所说的班级、性别、身高、年龄、考试分数、兴趣爱好、出生地址等，这些都是把学生进行归类分组的一个个维度。可能有些同学就要问了，这不就是分组吗？他跟分组有什么区别？乍看貌似 Facet 跟 Group 是一个概念，其实 Facet 跟 Group 还是有点区别的，比如，按考试分数统计，我们一般不会说统计 60 分有几人，61 分有几人，62 分有几人，63 分有几人……一直到 100 分，实际我们一般会这样统计：60 分以下有几人，60 ~ 70 分有几人，70 ~ 80 分有几人，80 ~ 90 分有几人，90 ~ 100 分有几人。这里说的 60 分以下、60 ~ 70 分、70 ~ 80、80 ~ 90、90 ~ 100 表示的是分数段，即数字范围，不是简简单单地按照某个域进行分组的，甚至会有更复杂的维度统计。比如，我要你统计各个分数段男生多少人、女生多少人，这其实就是多个查询条件组合成一个维度。说到这儿，我想大家应该都豁然开朗了，Facet 即维度不仅仅是建立在某个域上，它可以建立在对某个域进行函数计算后得到的计算值上，它还可以建立在某个查询条件上，该查询条件你可以任意组合，这是 Group 分组所办不到的。如果你仅仅是对某个域进行 Facet 统计，那就跟 Group 类似了。你可以把 Facet 理解为 Group 的火力升级版，功能更强大！

想要学好本章，你需要熟悉如何发起一个查询请求、Query Parser 是如何工作的，以及如何在 schema.xml 中定义域、域类型。最好还能够理解 Lucene 内部是如何存储索引的，但这不是必须的。

Facet 查询由两部分组成，即计算并显示 Facet 给用户和根据用户选定的值过滤结果集。此时，你可能想知道：到底 Facet 查询是什么？假如在手机 APP 上搜索外卖，你可能期望用户界面上能够提供多个维度作为导航元素方便你筛选外卖，正如图 6-1 所示。

在图 6-1 中，外卖类型、价格、区域就相当于 3 个 Facet，每个 Facet 下是相关的值列表以及各项匹配的索引文档的统计数字，当你单击其中任意一项，会返回该项下匹配的索引文档，这就是 Facet 查询。但是后面的统计数字并不是必需统计的，像淘宝、京东这些电商网站上的 Facet 查询部分仅仅只是罗列了每个 Facet 维度的子项，如图 6-2 所示。



外卖类型	价格	区域
快餐 (10073)	<20 (5674)	朝阳区 (2021)
特色菜系 (2530)	20~50 (1007)	丰台区 (1499)
异国料理 (1532)	50~100 (1301)	海淀区 (850)
小吃夜宵 (904)	100+ (556)	
甜食饮品 (409)		

图 6-1 查询界面上的导航元素



图 6-2 JD Facet 查询界面

为了方便大家更直观地理解 Facet，下面我会以搜索饭店为例子对 Facet 进行讲解。在开始之前，我们需要先导入一些测试数据到 Solr 中。

## 6.2 Facet 简单示例

首先我们需要创建一个 core，这里我们将 core 命名为 restaurants，然后按图 6-3 所示创建好配置目录、数据目录以及 lib 目录：

然后创建 core.properties 配置文件并稍作编辑，编辑内容如下所示：

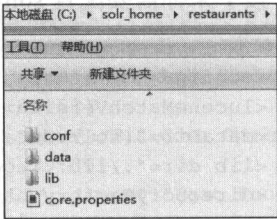


图 6-3 "restaurants" Core 目录结构

其中 name 表示我们的 core 名称，config 表示当前 core 的 solrconfig.xml 文件的加载路径，默认是相对于当前 core 的 conf 目录，同理 schema 表示我们当前 core 的 schema.xml 文件的加载路径，dataDir 表示我们的数据目录，默认是相对于当前 core 的根目录。然后我们需要在 conf 目录下配置我们的 schema.xml，如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="example" version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true" />
    <field name="_version_" type="long" indexed="true" stored="true"/>
    <field name="text" type="string" indexed="true" stored="false" multiValued=
```

```

"true"/>
  <field name="name" type="string" indexed="true" stored="true" />
  <field name="city" type="string" indexed="true" stored="true" />
  <field name="type" type="string" indexed="true" stored="true" />
  <field name="state" type="string" indexed="true" stored="true"/>
  <field name="tags" type="string" indexed="true" stored="true" multiValued=
"true" />
  <field name="price" type="double" indexed="true" stored="true" />
</fields>
<uniqueKey>id</uniqueKey>
<types>
  <fieldType name="boolean" class="solr.BoolField" sortMissingLast="true"/>
  <fieldType name="date" class="solr.TrieDateField" precisionStep="0" positionIncrement-
Gap="0"/>
  <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0" position-
IncrementGap="0"/>
  <fieldType name="long" class="solr.TrieLongField" precisionStep="0" position-
IncrementGap="0"/>
  <fieldType name="string" class="solr.StrField" sortMissingLast="true" />
</types>
</schema>

```

id 表示主键域；name 即饭店的名称；city 表示饭店所在城市；type 表示饭店的类型；state 表示饭店所在州；tags 表示体现饭店特色或优势的标签；price 表示饭店主营食物的价格。这里所有的域我们都定义成 indexed = true 且 stored = true，只是为了演示目的，实际项目中 stored 是不是设置为 true 根据需求决定。但 Facet 查询要求 Facet Field 必须是 indexed = true，如果一个域 indexed 不等于 true 那就无法在该域上进行查询，这点需要注意。接下来你需要配置 solrconfig.xml，配置示例如下所示：

```

<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <uceneMatchVersion>5.3.1</uceneMatchVersion>
  <dataDir>${solr.data.dir:}</dataDir>
  <lib dir="./lib" regex=".*\.jar"/>
  <directoryFactory name="DirectoryFactory"
class="${solr.directoryFactory:solr.NRTCachingDirectoryFactory}"/>
  <codecFactory class="solr.SchemaCodecFactory"/>
  <schemaFactory class="ClassicIndexSchemaFactory"/>
  <updateHandler class="solr.DirectUpdateHandler2">
    <updateLog>
      <str name="dir">${solr.ulong.dir:}</str>
    </updateLog>
    <autoCommit>
      <maxTime>15000</maxTime>
      <openSearcher>false</openSearcher>
    </autoCommit>
  </updateHandler>
  <query>
    <maxBooleanClauses>1024</maxBooleanClauses>
  </query>
</config>

```

```

<useColdSearcher>false</useColdSearcher>
<maxWarmingSearchers>1</maxWarmingSearchers>
</query>
<requestDispatcher handleSelect="false" >
<httpCaching never304="true" />
</requestDispatcher>
<requestHandler name="/update" class="solr.UpdateRequestHandler" />
<requestHandler name="/select" class="solr.SearchHandler">
<lst name="defaults">
<str name="echoParams">none</str>
<str name="df">text</str>
<str name="wt">json</str>
<str name="indent">true</str>
</lst>
</requestHandler>
<queryResponseWriter name="json" class="solr.JSONResponseWriter">
<str name="content-type">text/plain; charset=UTF-8</str>
</queryResponseWriter>
<admin>
<defaultQuery>*:*</defaultQuery>
</admin>
</config>

```

然后你需要往当前 core 的 lib 目录下添加依赖的 jar 包，具体需要导入哪些 jar 包请阅读第 1 章关于 Solr 安装部署的内容，这里不再赘述。此时我们就可以启动 Solr 使其能够加载新添加的 Core 啦！

下面是我们接下来执行 Facet 查询需要用到的测试数据：

```

[
  {"id":"1", "name":"Red Lobster", "city":"San Francisco, CA", "type":"Sit-
down Chain", "state":"California", "tags":["sea food", "sit down"], "price":33.00},
  {"id":"2", "name":"Red Lobster", "city":"Atlanta, GA", "type":"Sit-down Chain",
"state":"Georgia", "tags":["sea food", "sit-down"], "price":22.00},
  {"id":"3", "name":"Red Lobster", "city":"New York, NY", "type":"Sit-down Chain",
"state":"New York", "tags":["sea food", "sit-down"], "price":29.00},
  {"id":"4", "name":"McDonalds", "city":"San Francisco, CA", "type":"Fast Food",
"state":"California", "tags":["fast food", "hamburgers", "coffee", "wi-fi", "breakfast"],
"price":9.00},
  {"id":"5", "name":"McDonalds", "city":"Atlanta, GA", "type":"Fast Food", "state":
"Georgia", "tags":["fast food", "hamburgers", "coffee", "wi-fi", "breakfast"], "price":
4.00},
  {"id":"6", "name":"McDonalds", "city":"New York, NY", "type":"Fast Food",
"state":"New York", "tags":["fast food", "hamburgers", "coffee", "wi-fi", "breakfast"],
"price":4.00},
  {"id":"7", "name":"McDonalds", "city":"Chicago, IL", "type":"Fast Food", "state":
"Illinois", "tags":["fast food", "hamburgers", "coffee", "wi-fi", "breakfast"], "price":
4.00},
  {"id":"8", "name":"McDonalds", "city":"Austin, TX", "type":"Fast Food", "state":
"Texas", "tags":["fast food", "hamburgers", "coffee", "wi-fi", "breakfast"], "price":
4.00},

```

```

    {"id":"9", "name":"Pizza Hut", "city":"Atlanta, GA", "type":"Sit-down Chain",
"state":"Georgia", "tags":["pizza", "sit-down", "delivery"], "price":15.00},
    {"id":"10", "name":"Pizza Hut", "city":"New York, NY", "type":"Sit-down Chain",
"state":"New York", "tags":["pizza", "sit-down", "delivery"], "price":24.00},
    {"id":"11", "name":"Pizza Hut", "city":"Austin, TX", "type":"Sit-down Chain",
"state":"Texas", "tags":["pizza", "sit-down", "delivery"], "price":18.00},
    {"id":"12", "name":"Freddy's Pizza Shop", "city":"Los Angeles, CA", "type":"Local
Sit-down", "state":"California", "tags":["pizza", "pasta", "sit-down"], "price":
25.00},
    {"id":"13", "name":"The Iberian Pig", "city":"Atlanta, GA", "type":"Upscale",
"state":"Georgia", "tags":["spanish", "tapas", "sit-down", "upscale"], "price":
45.00},
    {"id":"14", "name":"Sprig", "city":"Atlanta, GA", "type":"Local Sit-down",
"state":"Georgia", "tags":["sit-down", "gluten-free", "southern cuisine"], "price":
15.00},
    {"id":"15", "name":"Starbucks", "city":"San Francisco, CA", "type":"Coffee
Shop", "state":"California", "tags":["coffee", "breakfast"], "price":7.50},
    {"id":"16", "name":"Starbucks", "city":"Atlanta, GA", "type":"Coffee Shop",
"state":"Georgia", "tags":["coffee", "breakfast"], "price":4.00},
    {"id":"17", "name":"Starbucks", "city":"New York, NY", "type":"Coffee Shop",
"state":"New York", "tags":["coffee", "breakfast"], "price":6.50},
    {"id":"18", "name":"Starbucks", "city":"Chicago, IL", "type":"Coffee Shop",
"state":"Illinois", "tags":["coffee", "breakfast"], "price":6.00},
    {"id":"19", "name":"Starbucks", "city":"Austin, TX", "type":"Coffee Shop",
"state":"Texas", "tags":["coffee", "breakfast"], "price":5.00},
    {"id":"20", "name":"Starbucks", "city":"Greenville, SC", "type":"Coffee Shop",
"state":"South Carolina", "tags":["coffee", "breakfast"], "price":3.00}
]

```

我们可以通过 Solr Web 后台的数据导入功能将测试数据导入到 Solr 中，如图 6-4 所示。



图 6-4 Solr Web 后台界面通过上传 JSON 文件方式导入数据

但是，在导入过程中，可能会出现如下这种异常：

```
Unsupported ContentType: application/octet-stream
Not in: [application/xml, text/csv, text/json, application/csv, application/
javabin, text/xml, application/json]
```

到底怎么回事儿？居然不能识别 JSON 文件的 MIME 类型，那么该如何解决这个问题呢？之所以出现这个异常，是因为 Solr 是完全依赖系统中注册的 MIME 类型来判断用户上传的文件的 MIME 类型，并不是简单的根据文件后缀名去判断文件类型以及 MIME 类型。因为我们当前操作系统中并没有注册 .json 文件对应的 MIME 类型，所以导致最终按照默认的 MIME 类型 application/octet-stream 来处理

如图 6-5 所示系统注册表下新建的 JSON 文件 MIME 类型。在 Windows 操作系统中，你需要打开系统注册表，添加一个键值对，具体操作为：



图 6-5 系统注册表下新建 JSON 文件的 MIME 类型

- 1) 在“运行”中输入 regedit 打开你的系统注册表；
- 2) 在 HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes 下新建→项：.json；
- 3) 在 .json 项下新建→字符串值：Content Type: application/json，冒号前面是 key，冒号后面是 value。

如果是在 Linux 操作系统上，你需要编辑 /etc/mime.types 文件，在其中添加如下内容：

```
application/json json
```

如果发现上传 XML 文件也出现同样的异常，那么你还需要再添加一行 application/xml xml。修改完成之后，你需要重启系统才能立即生效，Windows 操作系统同理。系统重启之后，再次上传 JSON 文件导入数据到 Solr 中创建索引就不会再报错了。导入成功之后通过 Solr Web 后台左侧的“Query”菜单查询下导入的索引文档总个数，如果能看到“numFound”：20 说明我们的 20 条测试数据已经导入成功了。

一切准备就绪，让我们开始 Facet 查询学习之旅吧。先从 Facet 查询最常见的方式开始：在指定域的每个唯一值上执行 Facet 查询即 Field Facet。

当你执行一个 Field Facet 查询时，它会返回该域上每个唯一值列表以及每个唯一值匹配的索引文档的总数。下面我们将学习如何创建一个 Field Facet 查询，以及学习 Facet 查询相关的请求参数。为了演示，在接下来的章节中，我们都将使用上一节中导入的索引数据进行 Facet 查询测试。下面是第一个 Facet 查询示例：

```
http://localhost:8080/solr/restaurants/select?q=*:*&rows=0&facet=true&facet.
field=name
```

下面是查询返回的结果：

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 180,
    "response": { "numFound": 20, "start": 0, "docs": [] }
  },
  "facet_counts": {
    "facet_queries": {},
    "facet_fields": {
      "name": [
        "Starbucks", 6,
        "McDonalds", 5,
        "Pizza Hut", 3,
        "Red Lobster", 3,
        "Freddy's Pizza Shop", 1,
        "Sprig", 1,
        "The Iberian Pig", 1
      ],
      "facet_dates": {},
      "facet_ranges": {},
      "facet_intervals": {},
      "facet_heatmaps": {}
    }
  }
}
```

上面示例中我们演示了 Solr 中 Facet 查询的最基本形式：在指定域上执行 Facet 查询，在这种 Facet 查询中，会对域的域值进行分组统计，当然如果该域配置了分词器，那么就是对域的域值分词后得到的每个唯一 Term 进行分组统计。

但并不是每个域的域值都是单个值，比如我们 schema.xml 中的 tags 域就是一个多值域，让我们看看在一个多值域上执行 Field Facet 查询会是什么情况？

[http://localhost:8080/solr/restaurants/select?q=\\*&facet=true&facet.field=tags](http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.field=tags)

下面是多值域 Facet 查询执行后返回的结果：

```
"facet_fields": {
  "tags": [
    "breakfast", 11,
    "coffee", 11,
    "sit-down", 8,
    "fast food", 5,
    "hamburgers", 5,
    "wi-fi", 5,
    "pizza", 4,
    "delivery", 3,
    "sea food", 3,
    "gluten-free", 1,
    "pasta", 1,
    "sit down", 1,
    "southern cuisine", 1,
    "spanish", 1,
    "tapas", 1,
    "upscale", 1
  ]
}
```



通过上面的查询结果，我们会发现：breakfast（早餐）和 coffee（咖啡）在 restaurant 测试文档中是出现频率比较高的两个。这是因为它们在测试文档中出现的频率比较高，而每个 Term 后面的统计数字就是它们在索引文档中出现的总次数。对于那些在索引文档中出现频率比较高的 Term 你可以通过 TagCloud（标签云）的形式展现给用户，比如出现频率高的 Term 使用大号字体，然后用户会有比较大的几率去单击比较醒目的 Term，从而实现导航用户的浏览行为。Tag Cloud（标签云）对于用户来说是从分类的角度来鸟瞰查询结果的常用方法。

除了可以使用示例中的 tags 多值域进行 Facet 查询之外，你还可以对纯文本域进行 Facet。一般纯文本域需要分词处理，但可能会包含一些噪音词，比如停用词之类的，也许会干扰你的 Facet 查询，因此你需要对文本域配置停用词过滤器。

另外一点需要记住的是 Facet 查询是针对域的唯一值进行分组统计，如果该域是 StringField，即不会进行分词处理，那么就直接根据该域的每个域值进行分组统计。但是对于 TextField，那么就是对分词后得到的每个 Token 进行分类统计，统计的数字就是每个 Token 在所有文档中出现的总次数。对于 Solr 开发者，我们一般建议单独建一个新域用于 Facet 查询，将域值复制到新的域，这里你可以使用 Solr 中的复制域。

此时你应该对 Facet 有个初步感觉了，对在单值域和多值域上执行 Facet 查询也基本熟悉了。目前还只是使用 Facet 的默认设置来执行查询。只对 20 个索引文档执行 Facet 查询看起来非常的容易，假如我们有上百万的 Term 呢？Solr 提供了很多 Facet 参数允许你对 Facet 查询进行调整或干预 Facet 查询行为。表 6-1 所示为可以在 Field Facet 中指定的参数。

表 6-1 可在 Field Facet 中指定的参数

参数	值	描 述
facet	true/false	表示对哪个域执行 Facet Search，此参数可以指定多次
facet.field	任意 indexed = true 的域	表示对哪个域执行 Facet Search，此参数可以指定多次
facet.sort	index/count	Count 表示按照统计数字大小排序，或者按照 term 的字母表顺序进行排序，默认值为 count
facet.limit	> = -1 的 Integer 数字	表示每个 facet 组中只返回前 N 项，此参数可以对每个 Facet 域进行设置
facet.mincount	> = 0 的 Integer 数字	表示统计的每一项数字的最小值，此参数可以对每个 Facet 域进行设置
facet.method	enum/fc/fcs	method = enum 时会遍历索引中的所有 Term，计算这些 Term 的交集。 method-fc（即 File Cache 的缩写）时会遍历匹配 Facet 查询的所有文档，然后找出包含该 Term 的文档。当域包含很多唯一值时使用 fc 执行速度会更快些，当域包含的唯一值只是几个，那么使用 enum 方式会更快些。对于所有域（Boolean 域除外），facet.method 默认值是 fc。fcs 会对单值 StringField 域提供基于每个段文件的 Field Cache（域缓存），当你的索引数据需要频繁更新时，对于 StringField 使用 fcs，执行性能会更好。它还可以接收一个本地参数，即通过指定处理不同段文件的线程数参数来加快 Facet 的执行速度。此参数可以对每个 Facet 域进行设置

(续)

参数	值	描 述
facet.enum.cache.minDf	>= 0 的 Integer 数字	高级参数：表示 Term 至少匹配多少个 Document 才启用域缓存，若此参数设置为 0，则表示始终都启用域缓存，默认值就是零。若此参数设置为大于零，则可以减少内存占用，但确是以减缓 Facet 查询速度为代价的。一般这个值推荐设置为 20 ~ 50，它并不影响最终的 Facet 返回结果，只是影响 Facet 查询性能。此参数可以对每个 Facet 域进行设置
facet.prefix	任意字符串	表示只对指定前缀的 Term 进行 Facet 查询，对于想要查找相似 Term 或用于构建 autocomplete 功能时会有用
facet.missing	true/false	表示是否对空值 Term 也进行 Facet 统计，默认值 false
facet.offset	>= 0 的 Integer 数字	用于指定 Facet 查询返回结果集的偏移量，偏移量从零开始计算。当你需要对 Facet 查询返回的结果集进行分页时会有用
facet.threads	Integer 数字	用于指定当对多个域执行 Facet 查询时使用的处理线程个数，默认值为零，表示默认情况下，会使用单线程串行的方式执行每个域上的 Facet 查询。此参数设置为正数，则表示会创建指定数量的线程以多线程并行的方式去执行 Facet 查询。设置为负数即表示线程数不受限制。当你有多个域需要执行 Facet 查询时，采用多线程方式明显会加快 Facet 查询速度。注意：此参数只适用于 Field Facet Query，并且当你的 Solr 查询并发量很大时请不要开启此参数，比如互联网电商项目中请不要使用

facet.field 参数可以指定多次，比如像下面这样：

```
&facet.field=tags&facet.field=type
```

有些 Facet 参数支持对每个域单独设置，基本语法如下所示：

```
f.<fieldName>.<FacetParameter>=<value>
```

其中 <fieldName> 表示域名称，<FacetParameter> 表示 Facet 参数名称，<value> 表示该参数对应的参数值。

### 6.3 Query Facet

除了能够对任意的索引域执行 Facet 查询之外，你还可以对任意的子查询统计匹配的索引文档总个数，而后你能够根据统计的数据进行分析。Solr 提供了这种功能，它被称为 Query Facet。

演示这种功能的最好方式就是通过示例。假如你想通过一个查询，统计出价格落在 [ 5, 25 ] 区间内的饭店，但同时你又想要了解美国东岸、西岸和中部各有多少饭店。你当然可以通过 4 个独立的查询来实现这个需求，如下所示：

```
// 查询价格落在 [5,25] 区间内的饭店
http://localhost:8080/solr/restaurants/select?q=*&fq=price:[5 TO 25]

// 过滤出查询价格落在 [5,25] 区间内的美国东部饭店个数
```

```
http://localhost:8080/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
fq=state:("New York" OR "Georgia" OR "South Carolina")
```

// 过滤出查询价格落在 [5,25] 区间内的美国西部饭店个数

```
http://localhost:8080/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
fq=state:("Illinois" OR "Texas")
```

// 过滤出查询价格落在 [5,25] 区间内的美国中部饭店个数

```
http://localhost:8080/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
fq=state:("California")
```

上面采用的方式通过 4 次独立查询虽然也能达到目的,但这种方式并不可取,下面我们将演示如何使用 Query Facet 将多个查询合并为一个查询。

```
http://localhost:8080/solr/restaurants/select?q=*&fq=price:[5 TO
25]&facet=true&
facet.query=state:("New York" OR "Georgia" OR "South Carolina")&
facet.query=state:("Illinois" OR "Texas")&
facet.query=state:("California")
```

查询返回的结果如下所示:

```
"facet_counts":{
  "facet_queries":{
    "state:(\"New York\" OR \"Georgia\" OR \"South Carolina\")":5,
    "state:(\"Illinois\" OR \"Texas\")":3,
    "state:(\"California\")":3},
```

正如你看到的那样,多个子查询可以通过 facet.query 参数结合成一个查询请求。同理,也可以将多个价格区间组合成一个 Query 请求,这样可以重构我们的查询请求,如下所示:

```
http://localhost:8080/solr/restaurants/select?q=*&rows=0&facet=true&
facet.query=price:[* TO 5]&
facet.query=price:[5 TO 10]&
facet.query=price:[10 TO 20]&
facet.query=price:[20 TO 50]&
facet.query=price:[50 TO *]
```

返回的查询结果如下所示:

```
"facet_counts":{
  "facet_queries":{
    "price:[* TO 5]":6,
    "price:[5 TO 10]":5,
    "price:[10 TO 20]":3,
    "price:[20 TO 50]":6,
    "price:[50 TO *]":0},
```

上面这个示例演示了如何在任意一个 Solr Query 中构建一个 Facet 查询并返回统计信息。其实,在上面示例中,你也可以创建一个 price\_range 域即表示当前 price 落在哪个区间,比

如 `price_range` 域的域值可以是这样的: `2 - 50`, 表示价格在 `[ 2, 50 ]` 之间, 这样你在添加索引文档时就需要确定每个 `price` 落在区间内, 然后赋值为 `price_range` 域并写入到索引中。

这样你就只需要根据 `price_range` 一个域进行 Facet 查询了, 但是这样做有个弊端, 即未来如果 `price` 域数据有变化, 那么 `price_range` 域也要随之更新, 你的索引需要重建。这是一个非常痛苦的过程, 特别是当你的索引文档数量在不断递增膨胀时。针对这个问题, Query Facet 提供了一个不错的选择。它允许在查询时非常灵活的指定或重新定义哪些 Facet Value 应该被计算统计和返回。

尽管我们的示例非常简单, 但它完全演示了基于任意查询构建 Facet Query 的灵活性, 因为 Solr 提供了很多强大的查询功能, 比如 Neseed Query (转换查询)、Function Query (函数查询)、Facet Query (维度查询)。想象一下, 可以借助 Facet Query 实现方圆  $N$  千米之内的地理位置区间范围 (比如 5 千米以内、 $5 \sim 10$  千米、 $10 \sim 20$  千米) 查询。你还可以通过一个自定义的相关性评分 Function 生成一个 Function Query, 而后基于 Function Query 计算结果值构建一个 Facet Query。你可以基于任何域、任何 Query 或动态计算值构建一个查询, 又可以基于任何查询构建一个 Facet 查询, 如何使用它完全靠你的想象力。

尽管 Query Facet 是极其灵活的, 但有时候让人不爽的是, 你需要显式指定每个想要基于它统计的值。为此, Solr 为 Facet Range Query (Facet 区间范围查询) 提供了便利, 它使得你在基于数字或日期值进行 Facet Range Query 时变得更加简单。

## 6.4 Range Facet

Range Facet, 顾名思义, 它表示 Facet 区间范围查询, 一般用于对数字或日期的区间范围查询。这类似于我们普通的区间范围查询, 但 Facet Query 会统计每个区间匹配的索引文档总数。以前当你有多个区间范围时可能需要指定 3 个 `facet.query` 参数来实现, 通过 Range Facet 可以简化你的查询。使用示例如下所示:

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&
facet.range=price&
facet.range.start=0&
facet.range.end=50&
facet.range.gap=5
```

上面示例中 `facet.range` 参数表示对哪个域执行 Facet 区间查询, `facet.range.start` 参数表示区间的上限值, `facet.range.end` 表示区间的下限值。 `facet.range.gap` 参数按照每个区间分布多少个值进行自动区间划分。

返回的查询结果如下所示:

```
"facet_ranges":{
  "price":{
    "counts":[
      "0,0",6,
```

```
"5.0",5,
"10.0",0,
"15.0",3,
"20.0",2,
"25.0",2,
"30.0",1,
"35.0",0,
"40.0",0,
"45.0",1],
"gap":5.0,
"start":0.0,
"end":50.0}},
```

首先 Range Query 会统计落入每个区间内的索引文档的总个数，尽管有些区间内并没有任何索引文档，但仍然返回了该区间统计信息。然后每个区间不再是人工去一个个指定了，而是通过指定一个 facet.range.gap 参数在限定的 facet.range.start 和 facet.range.end 参数之间自动进行区间分割。facet.range.gap 值越大，分割出的每个区间范围越大，gap 值如何设置取决于你的项目需求。通过这种自动分割区间的方式确实能够为我们节省很多时间，特别是区间范围个数很多的时候，你不用再手动指定 *N* 个 facet.query 参数。

这里还有其他的可选的参数，比如 facet.range.hardend、facet.range.other、facet.range.include。下面表 6-2 所示详细描述了每个参数的含义。

表 6-2 其他可选参数

参数	值	描 述
facet.range	任何 indexed = true 的数字域或 date 域的名称	指定你需要在哪个域上面执行 Facet Range Query，此参数可以指定多次
facet.range.start	数字域或 date 域的区间范围上限值即最小值	区间范围的上限值，此参数可以对每个 Facet 域进行设置
facet.range.end	数字域或 date 域的区间范围下限值即最大值	区间范围的下限值，此参数可以对每个 Facet 域进行设置
facet.range.gap	对于 date 域而言，gap 值可以为 DateMath 表达式，比如 + 1DAY、+ 2MONTHS、+ 1HOUR。 对于数字域，你只需要指定一个数字即可	gap 即表示区间递增的公差，用于按照指定的 gap（间隔）对 [facet.range.start, facet.range.end] 区间进行自动划分，生成多个子区间，此参数可以对每个 Facet 域进行设置
facet.range.hardend	true/false	表示含义还是举例说明吧，比如你 facet.date.start = 2015-01-01，而 facet.date.end = 2015-09-20，假如你 facet.date.gap = + 1MONTH 即表示按一个月把 start 与 end 之间的时间根据 gap 值分成 9 份，如果 hardend 为 true，那最后一份的时间范围是 2015-09-01 至 2015-09-20，如果 hardend 为 false，那最后一份的时间范围就是 2015-09-01 至 2015-10-01，即直接无视 end 参数的限制，严格按照 gap 的间隔来算，此参数可以对每个 Facet 域进行设置

(续)

参数	值	描 述
facet.range.other	before/after/between/all/none	before: 表示需要对 start 之前的日期做个统计; after: 表示需要对 end 之后的日期做个统计; between: 表示需要对 start 与 end 之间的日期做个统计; none: 表示不做任何汇总统计; all: 表示 before, after, between 都需要做统计, 此参数可以对每个 Facet 域进行设置
facet.range.include	lower/upper/edge/outer/all	lower: 表示划分的所有子区间都包含上限值即最小值; upper: 表示划分的所有子区间都包含下限值即最大值; edge: 表示划分的第一个子区间包含上限值即最小值, 最后一个子区间包含下限值即最大值; outer: 表示当你的 facet.range.other 参数设置为 before 或 after 时, 是否包含 before 和 after 这两个边界值; all: 表示分别指定上面 4 个参数 此参数可以指定多次, 且可以对每个域进行单独设置

对于 Field Facet, 你可以通过 `f.<fieldName>.<FacetParameter>=<value>` 语法为每个域指定 Range Facet 相关参数。

当你在对数字域或日期域进行区间范围的 Facet 查询时使用 Range Facet 语法显然要比使用普通的 Query Facet 更方便更简洁。当 Range Query 变得超级复杂时, 你可以选择使用多个独立的 Query Facet 来分解 Range Facet。在上面 3 种 Facet 查询中, Field Facet 是使用最广泛的一种, 而且它使用起来足够简单。

6.5 FacetFilter

基本上, 在 Facet 查询上应用一个 Filter Query 相比在 Query 上应用一个 Filter Query 并不难。假设在 3 个维度进行查询统计, 分别是 state 域、city 域以及根据 price 域的一个查询。那么你可以这样查询:

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]
```

返回的查询结果如下所示:

```
"facet_queries":{
  "price:[* TO 10]":11,
  "price:[10 TO 25]":5,
```



```

"price:[25 TO 50]":4,
"price:[50 TO *]":0},
"facet_fields":{
  "state":[
    "Georgia",6,
    "California",4,
    "New York",4,
    "Texas",3,
    "Illinois",2,
    "South Carolina",1],
  "city":[
    "Atlanta, GA",6,
    "New York, NY",4,
    "Austin, TX",3,
    "San Francisco, CA",3,
    "Chicago, IL",2,
    "Greenville, SC",1,
    "Los Angeles, CA",1]},

```

我们已经知道如何为一个普通 Query 添加 Filter Query，那么如果为 Facet 查询也添加一个 Filter 会发生什么呢？请看下面的示例：

```

http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]&
fq=state:California

```

上面查询中我们为 Range Facet Query 添加一个 Filter Query，最终返回的查询结果如下所示：

```

"facet_queries":{
  "price:[* TO 10]":2,
  "price:[10 TO 25]":0,
  "price:[25 TO 50]":2,
  "price:[50 TO *]":0},
"facet_fields":{
  "state":[
    "California",4],
  "city":[
    "San Francisco, CA",3,
    "Los Angeles, CA",1]},

```

上面示例中的 fq 参数表示的 Filter Query 作用于 state 域，同时对 Field Query 和 Query Facet 返回的结果集进行过滤，这点类似于普通 Query 上的 Filter Query。由于 Filter Query 会过滤掉不符合条件的索引文档，因此会影响每个 Facet 查询最终统计的数字。此外，你可

以为 Facet Query 添加多个 Filter Query，请看下面的查询示例：

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]&
fq=state:California&
fq=price:[* TO 10]
```

上面示例中我们为 Facet Query 应用了两个 Filter Query，对 Facet 查询返回的索引文档进行过滤，不符合条件的索引文档不进行 Facet 统计，最终返回的结果集如下所示：

```
"facet_counts":{
  "facet_queries":{
    "price:[* TO 10]":2,
    "price:[10 TO 25]":0,
    "price:[25 TO 50]":0,
    "price:[50 TO *]":0,
    "facet_fields":{
      "state":[
        "California",2,],
      "city":[
        "San Francisco, CA",2,],
      "facet_dates":{},
      "facet_ranges":{}}
```

正如你看到的那样，价格区间的 Facet 查询只有第一个区间有统计数字，其他区间由于都不符合 `fq = price: [* TO 10]` 这个条件，所以最终返回的索引文档总个数都为零。

我们的示例中的域都是单值域 `StringField`，因此如果单值域的域值不符合 Filter Query 的查询条件，那么就会导致对该域进行 Facet Query 时该域的某些值统计出来的数值为零，这对于单值域来说是有意义的。但是如果域是分词域（即域类型为 `TextField`）或者多值域（即 `multivalued = true` 的索引域，比如我们示例中的 `tags` 域）时，在这些域上执行 Facet 查询，会返回什么结果呢？那么，请看下面的示例：

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=name&
facet.field=tags
```

上面示例中，我们对 `name` 和 `tags` 域分别进行 Facet 查询统计，其中 `tags` 域是多值域，最终返回结果如下所示：

```
"facet_fields":{
  "name":[
    "Starbucks",6,
    "McDonalds",5,
```

```

    "Pizza Hut",3,
    "Red Lobster",3,
    "Freddy's Pizza Shop",1,
    "Sprig",1,
    "The Iberian Pig",1],
  "tags":[
    "breakfast",11,
    "coffee",11,
    "sit-down",8,
    "fast food",5,
    "hamburgers",5,
    "wi-fi",5,
    "pizza",4,
    "delivery",3,
    "sea food",3,
    "gluten-free",1,
    "pasta",1,
    "sit down",1,
    "southern cuisine",1,
    "spanish",1,
    "tapas",1,
    "upscale",1]],

```

你会发现多值域的值列表中的每一项都会单独统计，分词域同理。tags 域上 Facet 查询返回的项太多，同样可以为其添加 Filter Query 进行过滤，比如我们希望 tags 域上的 Facet 查询只返回包含 coffee 的索引文档的总个数，那么此时可以像下面这样执行查询：

```

http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&facet.
field=name&facet.field=tags&fq=tags:coffee

```

返回的查询结果如下所示：

```

"facet_fields":{
  "name":[
    "Starbucks",6,
    "McDonalds",5],
  "tags":[
    "breakfast",11,
    "coffee",11,
    "fast food",5,
    "hamburgers",5,
    "wi-fi",5]],

```

同理，你也可以为多值域的 Facet Query 添加多个 Filter Query，请看下面的查询示例：

```

http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&fq=tags:coffee&fq=tags:hamburgers

```

上面示例中，我们为 Facet Query 添加了两个 Filter Query，用于对 Facet Query 进行过滤，它只是影响最终每项返回的统计数字，并不会过滤掉某些统计项。最终返回的查询结果

如下所示：

```
"facet_fields":{
  "name":[
    "McDonalds",5],
  "tags":[
    "breakfast",5,
    "coffee",5,
    "fast food",5,
    "hamburgers",5,
    "wi-fi",5]}}
```

你会发现对于多值域，如果多值域的值列表中有部分不符合 Filter Query 的查询条件，那么符合查询条件的项仍然会被统计并返回的。

我们在上面示例中指定的每个 Filter Query 其实是毫无意义的。实际上，你可以将 `fq = tags: coffee&fq = tags: hamburgers` 改写成 `fq = tags: (coffee AND hamburgers)`。这种写法会减少在 Filter 缓存中查找的次数，而且也更容易控制多个 Filter Value 之间的交互。完全没必要在 Facet Query 中使用 AND 操作符去连接多个 fq。

前面的示例中我们大都是在对单值域进行 Facet Query，但实际上，通常更具挑战性的是我们需要对多值域或分词域进行 Facet Query。举个例子，假如在一个分词域上进行 Facet Query，其中有一个域值为 Los Angeles，经过分词后生成两个 token，即 Los、Angeles（假设不考虑大小写转换和词干还原），若我们的 Filter Query 表达式是 `fq = city: Los Angeles`，此时实际含义是查询包含 Los、Angeles 的索引文档，然后统计符合要求的索引文档总个数。为了在 fq 中支持使用空格分割的多个 Term 的 Phrase Query，你需要将多个 Term 使用双引号包裹起来。因此此时的查询语法就是：

```
fq=city:"Los Angeles"
```

当你盲目地对多个 Term 使用双引号进行包裹，为 Facet Query 添加 Filter Query 时，可能会遇到问题。如果使用双引号包裹多个 Term，那么查询语法会被破坏，除非你对查询表达式中特殊字符进行转义。因此，如果你的查询表达式中已经包含了双引号，那么此时需要将查询表达式中的双引号转义成 `\`，比如 `fq = name: "the \"in\" crowd"`。双引号和转义双引号已经够让你心烦了，但是还需要小心谨慎的是 Solr 对于分词域的处理后生成了哪些 Token，因为后续对该分词域进行 Facet Query 时，需要对分出来的每个 Token 进行查询统计，你需要在脑海中清楚内部这种分词行为是如何工作的以及最终会生成什么，而且分词还分索引阶段和查询阶段，这又为你增加了点复杂度。庆幸的是，Solr 已经为我们提供了 TermQParserPlugin 来解决此类问题，比如你可以使用 `fq = {!term}the "in" crowd` 来构建一个 TermQuery，此时不需要考虑双引号转义问题。

使用 Term Query Parser 来构建 Facet Query 上的 Filter Query 的一个缺点就是它不支持 Boolean 语法，所以如果你想在在一个 Filter Query 里连接多个 Facet 值，可能需要使用 Nested

Query Parser 来实现，下面分别演示两个方式连接你的 Filter Query：一种就是通过 Term Query Parser 定义多个 Filter Query；另一种就是通过 Nested Query Parser 连接两个 Term Query，而后在 Filter Query 中通过 AND 操作符连接两个 `_query_` (Nested Query 即嵌套查询，它能够将任意查询转换成 `_query_` 这个“伪域”)。

```
// 定义多个 Filter Query
http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&
fq={!term f=tags}coffee&fq={!term f=tags}hamburgers
// 通过 Nested Query Parser 定义多个伪域 _query_，然后在一个 Filter Query 中通过 AND 操作符
连接两个“伪域”
http://localhost:8080/solr/restaurants/select?q=*&
facet=true&facet.mincount=1&facet.field=name&facet.field=tags&
fq=_query_:"{!term f=tags}coffee" AND _query_:"{!term f=tags}hamburgers"
```

在本节中，你已经学会了如何在一个 Facet Query 上添加 Filter Query，它跟在普通 Query 上添加 Filter Query 没什么太大差别。你应该也知道如何为每个 Facet 值分别添加一个独立的 Filter Query 或者为多个 Facet 值添加一个 Filter Query。除此之外，你应该也知道了你可以使用 Term Query Parser 来忽略文本处理，以避免非常难处理的特殊字符转义问题。此时，你应该知道如何使用各种 Facet Query（比如 Field Facet、Query Facet、Range Facet、Facet Filter）去实现各种查询统计。但是 Facet 学习仍未结束，还有一些内容我们尚未覆盖到，比如出于友好显示目的为 Facet 重命名，以及已经被 Filter Query 过滤掉的索引文档如何也纳入 Facet 统计。

## 6.6 Multiselect Faceting

当我们发起一个 Facet Query，Facet 返回 Facet 名称可能并不是我们想要的，为此，Solr 允许你在 Facet Query 结果返回之前修改 Facet 的显示名称，以更友好的名称返回给用户。对于那些已经被 Filter Query 过滤掉的 Document，Solr 也允许将其纳入 Facet 统计之内，Solr 中将这种功能称为 Multiselect Faceting。即 Filter Query 能过滤查询结果集中将要返回的索引文档，但并不影响最终统计的索引文档总个数，看起来就像 Filter Query 并没有起作用一样。在本节中，我们将介绍关于 key、tag 以及 exclude 的概念和使用，它们启用了 Facet 中非常有用的重命名以及多选功能。

### 6.6.1 key

所有的 Facet（维度）都有一个方便开发者区分它们彼此的名称，如果是 Field Facet 或 Range Facet，那么 Facet 的名称就是 Field 的名称，如果是 Query Facet，那么 Facet 的名称就是 query 的查询表达式或者 Facet Value 或 Function 动态计算的值。通过使用 key 这个本

地参数, 你可以很容易的对任何 Facet 名称进行重命名, 具体请看下面的示例:

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field={!key="Location"}city&
facet.query={!key="<$10"}price:[* TO 10]&
facet.query={!key="$10 - $25"}price:[10 TO 25]&
facet.query={!key="$25 - $50"}price:[25 TO 50]&
facet.query={!key=">$50"}price:[50 TO *]
```

最终返回的查询结果如下所示:

```
"facet_queries":{
  "<$10":11,
  "$10 - $25":5,
  "$25 - $50":4,
  ">$50":0},
"facet_fields":{
  "Location":[
    "Atlanta, GA",6,
    "New York, NY",4,
    "Austin, TX",3,
    "San Francisco, CA",3,
    "Chicago, IL",2,
    "Greenville, SC",1,
    "Los Angeles, CA",1]],
```

Facet 的重命名功能在很多场景下非常有用, 它允许你的搜索程序请求 Query Facet。举个例子, 你不需要在接收到 Facet Query 返回的结果集之后的处理阶段去理解底层的 Query, Facet 返回的结果集会以比较友好的名称显示每项统计数据, 而你也不用关心当前 Facet Query 查询在底层是基于哪个域或者关联哪个 Query。此外可以为每个 Facet 定义一个唯一的名称, 通过指定 key 参数的方式可以为同一个域指定多个唯一的名称, 这对于 Field Facet 和 Range Facet 来说会比较有用。同时多个域也可以映射到同一个名称上, 不过这依赖于你查询时定义的查询条件。

假设索引文档中有个名称为 SecretInformationOnlyAvailableToSomeUsers 的域以及名称为 InformationAvailableToAllUsers 的域, 你可以在查询时构建这样一个 Facet Query:

```
facet.field=
{!key="Information "}InformationAvailableToAllUsers or
facet.field={!key=
"Information "}SecretInformationOnlyAvailableToSomeUsers.
```

key 参数带来的 Facet 名称重命名功能有多么便利就不言自明了。此外 Solr 中提供了 tag 参数, 它能够为某些 Facet 中的 Filter Query 打上标签, 以便于你能够控制它与 Solr 其他功能之间的交互。

## 6.6.2 tag

当一个 Filter Query 应用到一个 Solr Query 请求上时, 最终返回的结果集是求 Query 查



询返回的结果集与 Filter Query 查询返回的结果集之间的交集。默认情况下, Facet Query 也是这种查询机制。不符合 Filter Query 的 Facet Value 已经被排除在外, 不会纳入 Facet 查询统计, 这在大多数情况下是有用的。

但是这种查询机制仍然存在问题, 比如你想查询统计“California”州下的饭店数量, 你可能会应用一个 Filter Query: `fq = state: California`。此时其他州的饭店数量就不会统计了, 如果你仍然希望能够统计其他州的数据, 以便于继续单击其他州继续你的浏览查询行为, 而不是通过浏览器返回到上一级页面, 那么你就需要使用 Facet Exclusion 功能。Facet Exclusion 功能允许你将那些已经被应用在域 Facet Query 之上的任意 Filter Query 移除掉的索引文档重新添加到 Facet 查询统计中。这样在统计 Facet 数量的时候可以完全忽略任意 Filter Query 的影响, 而应用于 Facet Query 之上的 Filter Query 将只会影响最终返回的索引文档, 但并不影响每个 Facet 统计的文档总个数。实现这种功能你需要 `tag` 和 `ex` 这两个本地参数。下面的示例演示如何使用这两个参数实现 Multiselect Faceting。

```
http://localhost:8080/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field={!ex=tagForState}state&
facet.field={!ex=tagForCity}city&
facet.query={!ex=tagForPrice}price:[* TO 5]&
facet.query={!ex=tagForPrice}price:[5 TO 10]&
facet.query={!ex=tagForPrice}price:[10 TO 20]&
facet.query={!ex=tagForPrice}price:[20 TO 50]&
facet.query={!ex=tagForPrice}price:[50 TO *]&
fq={!tag="tagForState"}state:California
```

上面的示例中, 重点就是我们为 Facet Query 定义了一个 Filter Query, 即过滤掉不在 California 州的饭店, 同时通过 `tag` 参数为 Filter Query 打上了一个标签, 名为 `tagForState`, 标签名称是可以随意取的。然后在每个 Query Facet 上通过 `ex` 参数来应用我们刚刚打的那个标签, 应用一个标签的隐含的含义就是当前 Facet Query 自动忽略该标签指代的 Filter Query 对索引文档总个数统计阶段的影响, 但是 Facet Query 最终返回的索引文档仍然会进行过滤。其中 `ex` 即 `exclude` 的缩写即排除的意思。最终返回的查询结果如下所示:

```
"facet_counts":{
  "facet_queries":{
    "{!ex=tagForPrice}price:[* TO 5]":0,
    "{!ex=tagForPrice}price:[5 TO 10]":2,
    "{!ex=tagForPrice}price:[10 TO 20]":0,
    "{!ex=tagForPrice}price:[20 TO 50]":2,
    "{!ex=tagForPrice}price:[50 TO *]":0,
  },
  "facet_fields":{
    "state":[
      "Georgia",6,
      "California",4,
      "New York",4,
      "Texas",3,
```

```
"Illinois",2,
"South Carolina",1},
"city":{
"San Francisco, CA",3,
"Los Angeles, CA",1}},
```

正如上面 Facet Query 返回的统计数字显示的那样, Georgia=6 即表示 Georgia 州下有 6 家饭店, 并没有受 Filter Query 过滤条件的影响, 其他州的统计数据依然正常返回, 但是查看返回的索引文档部分, 即 JSON 字符串中的 docs 部分:

```
"docs":[
{
  "id":"1",
  "name":"Red Lobster",
  "city":"San Francisco, CA",
  "type":"Sit-down Chain",
  "state":"California",
  "tags":["sea food",
        "sit down"],
  "price":33.0,
  "_version_":1546855374529757184},
.....// 其他省略
```

你会发现, 最终返回的索引文档却又全部是 Carolina 州下的饭店信息, 跟 Filter Query 的过滤条件相吻合。

那么设计这种功能到底有什么用呢? 或者说到底什么场景下可以使用到它呢? 我想这是大家此刻最想知道的事情。下面依然以搜“饭店”为例进行说明, 假设你进入一个饭店的搜索界面, 界面初始在网页左侧或者其他位置显示了每个州下饭店的数量, 当然可能还有其他 Facet (维度) 的统计, 比如价格区间, 这里暂且以州这个维度进行讲解说明。看到下面这个界面展示, 你可以很清楚的了解到每个州下有多少饭店:

```
Georgia(6)
California(4)
New York(4)
Texas(3)
Illinois(2)
```

假设随机单击了 Georgia 州这个链接, 那么此刻后台的搜索程序会将你单击的这个 Georgia 作为一个过滤条件, 过滤出仅仅在 Georgia 州下的饭店, 底层构建的 Filter Query 可能是类似这样的: fq = state: Georgia, 正常情况下会在网站界面的右侧为你展示所有 Georgia 州下的饭店, 但默认情况下 Filter Query 还会影响最终 Facet Query 的对每个 Facet (维度) 下匹配的索引文档总个数的统计。即在你单击了 Georgia 州这个连接之后, 网页左侧的 Facet 统计数据展示部分可能会变成下面这样:

```
Georgia(6)
```

其他州的统计数据不显示了，因为它们不符合 `fq = state: Georgia` 这个过滤条件已经被 Filter Query 排除掉了。此时，这并不是我们想要的结果，因为可能你还想要再浏览看看 California 州有哪些饭店，怎么办？没办法，你只能通过浏览器后退返回上一级重新选择单击，这是一个很不爽的用户体验。但是如果当我们单击某个州的链接，能够在右侧展示符合该过滤条件下的所有饭店，但左侧的 Facet 统计不受影响，那么就可以实现连续单击浏览行为了，即此刻我们希望在单击了某个州链接之后，左侧的 Facet 统计数据展示部分保持不变，即依然如下显示：

```
Georgia(6)
California(4)
New York(4)
Texas(3)
Illinois(2)
```

那么此时你就需要结合 `tag + ex` 这两个本地参数来实现这个功能。

在 Solr 中，Facet 查询大量使用了 Solr 缓存，所以如果想要最大化的提升 Facet 查询性能的话，那么你需要优化 Solr 的内置缓存。关于 Solr 缓存将在后续章节详细讲解。除了 Solr 性能调优，你可能还希望了解 Facet 更高级方面的知识，其中一个就是 Pivot Faceting。Solr 提供了两个 Facet 叠加分组统计的功能类似 SQL 里的 Group by 两个字段的含义。即在某个 Facet Query 执行后返回的结果集基础之上再执行其他 Facet Query。如果没有 Pivot Faceting，要实现这种功能，你可能需要执行多次独立的 Facet Query，不过这种做法它没有很好的系统伸缩性，而且当索引文档数据量不断增长的时候，它很容易导致你必须运行几十甚至几百个 Facet 查询，这显然不可取，Solr 中的 Pivot Faceting 就是设计用来解决此类问题的。Pivot Faceting 允许你跨多个维度在一个查询中进行 Facet 查询统计。有关 Pivot Facet 留到后续章节再做讲解。

## 6.7 本章总结

Facet 提供了一种快速的方式使用户能够鸟瞰查询匹配的各种索引文档。你已经很难找到一个以搜索为准的网站没有提供 Facet 功能了。通过 Facet 你可以针对指定域进行 Facet（即 Field Facet），针对任意 Query 进行 Facet（即 QueryFacet），针对指定域的给定的区间进行 Facet 即 Range Facet。你还可以使用 Filter Query 对 Facet Query 进行过滤即 Facet Filter。如果想排除 Filter Query 对 Facet Count 的影响，那么你可以使用 `tag` 和 `ex` 参数来实现。最后可能还需要关注 Solr 提供的多维度叠加的 Facet 功能（即 Pivot Faceting）。此时此刻你应该已经能够实现一些比较复杂的查询。下一章我们将要学习如何使用 Solr 中另一个比较常见的功能：Highlighting（即高亮），它允许将匹配的索引文档里包含的搜索关键词高亮突出显示出来。

## Solr 高亮

通过第7章，你将可以学习到如下内容：

- ❑ 理解什么是 Solr 高亮；
- ❑ 熟悉 Solr 高亮工作原理；
- ❑ Solr 高亮的简单使用；
- ❑ 如何使用 FastVectorHighlighter 高亮器；
- ❑ 如何使用 PostingHighlighter 高亮器。

在本章中，我们将继续介绍 Solr 的另一个核心功能：Highlighting（高亮），即在查询返回的结果中高亮显示命中的查询关键字。Solr 高亮能够帮助用户快速地从查询结果集中扫描出哪些结果值得进一步的查阅浏览，或者决定是否应该单击进入下一页，甚至在发现匹配不到期望的结果信息时重新发起另一个查询请求。

### 7.1 什么是 Solr 高亮

Google 我想大家应该都用过，比如当你想要学习 Solr Highlighting（高亮）相关知识，你可能会在搜索输入框中输入如图 7-1 所示的搜索关键字：

然后单击“Google 搜索”，Google 搜索引擎会返回给你与该搜索关键字相关的结果，如图 7-2 所示。

返回的每条查询结果都由三部分组成，从上至下依次显示的是标题、网页的 URL、高亮摘要。你会发现，在高亮摘要部分会将用户输入的搜索关键词匹配的词进行高亮显示，这里 Google 是采用红色字体进行突出显示。高亮效果的展现形式取决于你，可以加粗、改

变字体颜色、更换其他字体等。摘要中高亮显示的关键词个数能显著地提示用户该条搜索结果的匹配相关度如何，比如高亮摘要中高亮显示的关键词个数越多则表明该条搜索结果越符合你的期望。但在我们上面的截图中，则是一个例外。虽然我们的第二条搜索结果完全匹配了我们在搜索时输入的搜索关键字，但即使第一条也匹配了两个关键字，匹配的顺序却跟我们输入的顺序是颠倒的。然而这并不影响它排在首位显示，这主要是因为考虑第一条是 Solr 官方提供的内容，更具有权威性，故优先展示。一句话，在返回的结果界面里，对命中的搜索关键字进行高亮标注出来，这就是 Solr 中的高亮功能，英文术语称为：Solr Highlighting。



图 7-1 使用 Google 搜索 “solr highlighting”

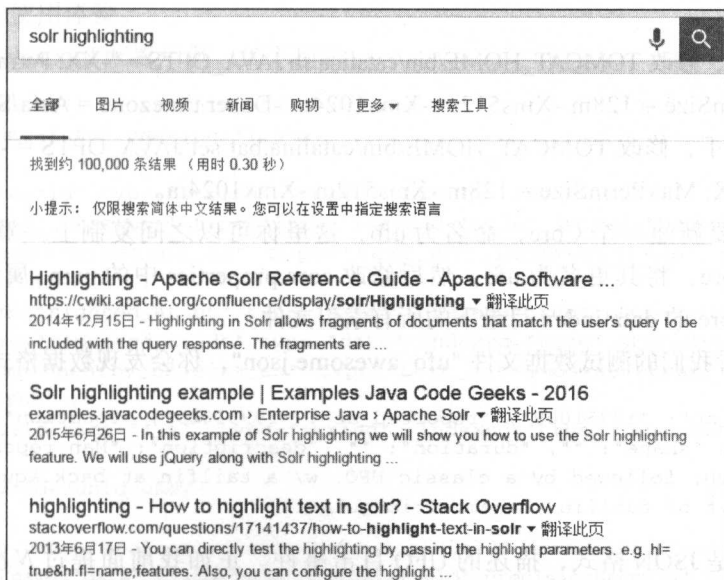


图 7-2 Google 搜索 “solr highlighting” 返回的结果页面

但需要注意的是，高亮摘要并不是索引文档中的 description 描述域，它只是根据用户

输入的搜索关键字以及描述域在查询时动态计算出来的一段文本摘要，正因为高亮摘要是动态计算的，所以它并不是一开始就存在的，而且它对于每个索引文档来说也不是固定不变的，会跟随用户输入的搜索关键字的不同发生动态变化。

在大部分的搜索应用程序中，由于显示屏幕空间有限，你不可能完全显示某一条结果。当然如果你的索引文档都是很短的文档，完全显示自然没有太大的问题。但是大多数情况下，你只能显示每个索引文档的一小部分。那这就会有问题了，你会选择索引文档中的哪部分内容用于显示呢？理论上来讲，你期望显示每个索引文档的一小片段，而该一小片段是基于与用户查询的相关度来动态生成与用户给定查询匹配度最高的最佳片段。而该片段也被称为“高亮摘要”，将动态生成最佳的摘要返回给用户也是 Solr 中的 Highlighting（高亮）提供的核心功能。

## 7.2 Solr 高亮的工作原理

开始之前，我们需要导入一些测试数据并搭建一个高亮的示例程序。第一步，你需要导入测试数据到 Solr 中建立索引。测试数据你可以从我的 Github 上获取。从我的 Github 上下载 ufo\_awesome.json 测试数据文件，该文件里包含了 6 万条测试数据。紧接着需要将其导入到 Solr 中。由于数据量有点大，为了防止在导入过程中出现 OOM（OutOfMemory）异常，这里建议你设置 head 大小最小为 512M。关于如何对 Tomcat 进行内存优化，请看下面这两段说明：

❑ Linux 下，修改 TOMCAT\_HOME/bin/catalina.sh `JAVA_OPTS="-XX:PermSize=64M-XX:MaxPermSize=128m-Xms512m-Xmx1024m-Duser.timezone=Asia/Shanghai"`；

❑ Windows 下，修改 TOMCAT\_HOME/bin/catalina.bat `set JAVA_OPTS=-XX:PermSize=64M-XX:MaxPermSize=128m-Xms512m-Xmx1024m`。

然后你需要新建一个 Core，命名为 ufo，这里你可以之间复制上一章中我们创建的 "restaurants" Core，将其更名为 ufo，然后修改 core.properties 中的 name 属性值为 ufo。然后清空 "ufo" Core 的 data/index 目录下的所有索引文件。

稍微看一看我们的测试数据文件 "ufo\_awesome.json"，你会发现数据格式大致如下：

```
{"sighted_at": "19951009", "reported_at": "19951009", "location": "Iowa City, IA", "shape": "", "duration": "", "description": "Man repts. witnessing &quot;flash, followed by a classic UFO, w/ a tailfin at back.&quot; Red color on top half of tailfin. Became triangular."}
```

测试数据是 JSON 格式，描述的 UFO 目击事件。正如我前面提过 *N* 次的，Solr 支持对 JSON 格式的数据进行导入。上一章中我们已经演示了如何通过 Solr 的 Web 后台采用上传 JSON 文件方式导入数据到 Solr 中。但导入之前，我们需要在 schema.xml 中定义 field。表 7-1 中展示了我们的示例中应该定义哪些域。





表 7-1 在 schema.xml 中定义域示例

域名称	域类型	描 述	示例数据
id	string	根据其他域手动生成的一个唯一标识值作为主键域。	20041130/20041204/...
sighted_at_dt	date	UFO 事件被目击到的时间。	2004-11-30T07:00:00Z
reported_at_dt	date	UFO 事件被报道的时间。	2004-12-04T07:00:00Z
month_s	string	UFO 事件发生的月份。	November
city_s	string	UFO 事件发生在美国的哪个城市。	Lancaster
state_s	string	UFO 事件发生在美国的哪个州。	OH
location_s	string	UFO 事件发生在美国的哪个州的哪个城市，location 是 JSON 数据的原始属性，这里添加 location_s 域是为了 Facet 查询测试。	Lancaster, OH
shape_s	string	UFO 的形状。	fireball
duration_s	string	UFO 出现了多久。	5 seconds
sighting_en	text_en	关于 UFO 事件的一些描述信息。	Bright blue fireball in the distance during a rainstorm.

这里我们使用了动态域，对于 string 类型的域我们为其域名添加了 \_s 后缀，对于 date 类型的域我们为其域名添加了 \_dt 后缀。这里我们还额外了添加 month\_s 域用于 facet 查询，比如用户可能想要根据 UFO 事件发生的月份还查询特定某个月的发生的 UFO 事件，再比如用户可能会想要了解“上个月发生的 UFO 时间”。但添加 month\_s 域并不是必需的，只是为了提升用户的使用体验。

下面是 schema.xml 配置示例，如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="ufo" version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true" />
    <field name="_version_" type="long" indexed="true" stored="true"/>
    <field name="sighting_en" type="text_en" indexed="true" stored="true"
multiValued="true" />
    <field name="sighted_at_dt" type="date" indexed="true" stored="true"/>
    <field name="reported_at_dt" type="date" indexed="true" stored="true"/>
    <field name="nearby_objects_en" type="text_en" indexed="true" stored="true"
multiValued="true" />
    <dynamicField name="*_s" type="string" indexed="true" stored="true" />
  </fields>
  <uniqueKey>id</uniqueKey>
  <types>
    <fieldType name="boolean" class="solr.BoolField" sortMissingLast="true"/>
    <fieldType name="date" class="solr.TrieDateField" precisionStep="0" positionIncrement
Gap="0"/>
    <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0" position
IncrementGap="0"/>
    <fieldType name="long" class="solr.TrieLongField" precisionStep="0" position
```

```
IncrementGap="0"/>
<fieldType name="string" class="solr.StrField" sortMissingLast="true" />
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
<analyzer type="index">
<tokenizer class="solr.StandardTokenizerFactory"/>
<filter class="solr.StopFilterFactory"
ignoreCase="true" words="stopwords_en.txt" />
<filter class="solr.LowerCaseFilterFactory"/>
<filter class="solr.EnglishPossessiveFilterFactory"/>
<filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
<filter class="solr.PorterStemFilterFactory"/>
</analyzer>
</fieldType>
</types>
</schema>
```

上面 text\_en 域类型定义中需要加载 stopwords\_en.txt 和 protwords.txt 两个字典文件，你需要将它们放置在跟 schema.xml 文件同级目录下，stopwords\_en.txt 字典文件中定义的是些英文中的停用词，protwords.txt 字典文件中定义的是需要保持原型不进行词干还原的单词。至于 solrconfig.xml，与上一章的 "restaurants" Core 中的 solrconfig.xml 配置保持一致即可。配置文件搞定后，你需要重启你的 Tomcat 使其能够自动发现我们新建的 "ufo" Core。

最后你可以通过 Solr 的 Web 后台上传测试数据文件 ufo\_awesome.json 导入数据，这里也可以通过 SolrJ 编程的方式导入数据。通过 SolrJ 方式导入数据到 Solr 中的示例代码请到我的 Github 上下载源码，下载到源码后直接运行 IndexUFO 类也能完成数据的导入。关于 SolrJ 的详细使用会留到后面的章节做详细讲解，这里暂不展开。

此时想必你已经完成了高亮测试数据的导入，下面开始 Solr 高亮学习。在 Solr 中使用高亮不需要做太多预先的配置工作，它是开箱即用的。高亮是 Solr 中的一个核心功能，通常也是大部分搜索程序中的一个重要功能。在本章中，你将会学习到如何启用 Solr 中的高亮功能，以及如何控制为搜索结果中的每个索引文档创建的高亮片段的个数。下面的代码清单演示了如何在 Solr 中开启高亮功能：

```
http://localhost:8080/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&wt=json&rows=10&hl=true
```

返回的高亮部分结果如下所示：

```
"highlighting":{
"20041130/20041204/lancaster/oh/fireball/2bbc6dc90efcbb8fb8f54ba23e07bd0a":{
"sighting_en":["Bright <em>blue</em><em>fireball</em> in the distance during
a <em>rain</em> storm. Not a lightning storm, no thunder"]},
// ..... 其他省略
```

Solr 返回的高亮部分内容在 “highlighting” 属性中，不难发现返回的高亮片段中 blue 和 fireball 都与搜索关键字匹配且两者在索引中也是紧紧挨在一起的，理论上来讲，两者应该使用一个高亮标签进行高亮标注，然而实际却是使用两对 <em></em> 元素来标注，如果

想要两者一起标注,则需要使用 `PostingsHighlighter` 高亮器。

在 Solr 中开启高亮功能,你只需要添加一个 `hl = true` 参数即可。同 Solr 中其他搜索组件类似, Solr 中的高亮器也支持一些可选的请求参数,用于调整高亮器的行为。让我们来使用这些可选参数来感受下它的作用。

在大部分情况下,每个结果对应一个高亮片段,对于用户来说通过高亮片段的个数来判断索引文档是否值得浏览可能还不够。举个例子,比如上面示例中返回的第二个文档对应的高亮片段中:

```
Horizontal, no arc. Fast. Huge. I described it to a friend who said it was a
"Blue Fireball"
```

乍一看,这个结果很符合要求,因为它的描述信息中提到了我们的搜索关键字“blue fireball”,但它只是排在第二位,说明它的相关性还不是很,下面我们添加一个 `hl.snippets` 参数,让搜索结果中的每个索引文档都返回多个高亮片段,请看下面的查询示例:

```
http://localhost:8080/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&wt=json&hl=true&hl.snippets=2
```

最终返回的高亮结果如下所示:

```
"highlighting":{
  "20041130/20041204/lancaster/oh/fireball/2bbc6dc90efcbb8fb8f54ba23e07bd0a":{
    "sighting_en":["Bright <em>blue</em><em>fireball</em> in the distance during a
<em>rain</em> storm. Not a lightning storm, no thunder"]},
    "20051110/20051116/lakeoswego/or/cylinder/4bae25d796ba82677ea0d77b36d08faf":{
      "sighting_en":[". Horizontal, no arc. Fast. Huge. I described it to a friend
who said it was a \"<em>Blue</em><em>Fireball</em>\",
        "Brilliant <em>blue</em> oblong object zooms horizontally across southern
sky at 2 in the morning. I awoke"]},
```

你会发现返回的高亮结果中有的索引文档有 1 个高亮片段,而有的却有 2 个高亮片断,从这点就能看出哪个索引文档相关度更高,默认最多返回一个高亮片段。

从返回的高亮片段结果来看,我们不难发现,高亮片段其实也是按照与搜索关键字的相关性评分来排序决定最后的显示先后顺序的。但是将 `hl.snippets` 设置为 2 并不能保证每个索引文档最终一定会生成 2 个高亮片段,比如我们高亮结果中的第一个 Document 就只有 1 个高亮片段尽管 `hl.snippets = 2`, `hl.snippets` 参数只是限制每个索引文档最终返回的高亮片段最多有几个,但此参数还受其他参数的影响,所以此参数并不是强制性要求。

OK,现在你已经感受到了高亮功能的作用,并且也动手去实践实现了 Solr 中的高亮,对 Solr 中的高亮也有了初步了解。下面我们将深入去了解 Solr 高亮背后的一些细节。试想下,为什么我们仅仅添加一个 `hl = true` 参数,高亮功能就开启了呢?还记得我们在 `solrconfig.xml` 中配置的查询组件吗?不记得也没关系,那么就现在打开 `solrconfig.xml`,你会看到这里配置了一个 `SearchHandler` 查询请求处理器,它监听了 `/select` 这个请求 URL,而

solr.SearchHandler 底层对应的 org.apache.solr.handler.component.SearchHandler 这个类，该类会在 inform 方法内部调用 getDefaultComponents 方法来初始化默认的查询组件，如图 7-3 所示。

```
protected List<String> getDefaultComponents()
{
    ArrayList<String> names = new ArrayList<>(6);
    names.add( QueryComponent.COMPONENT_NAME );
    names.add( FacetComponent.COMPONENT_NAME );
    names.add( FacetModule.COMPONENT_NAME );
    names.add( MoreLikeThisComponent.COMPONENT_NAME );
    names.add( HighlightComponent.COMPONENT_NAME );
    names.add( StatsComponent.COMPONENT_NAME );
    names.add( DebugComponent.COMPONENT_NAME );
    names.add( ExpandComponent.COMPONENT_NAME );
    return names;
}
```

图 7-3 SearchHandler 的内置查询组件请求链中默认包含的查询组件

从上面的代码，我们可得知，在 Solr 的 SearchHandler 初始化时，Solr 默认就帮我们初始化了上图所示的 8 种查询组件，HighlightComponent 高亮查询组件就是查询组件链条的一个组成部分。

图 7-4 形象地解释了 Solr 中的查询组件是如何工作的。

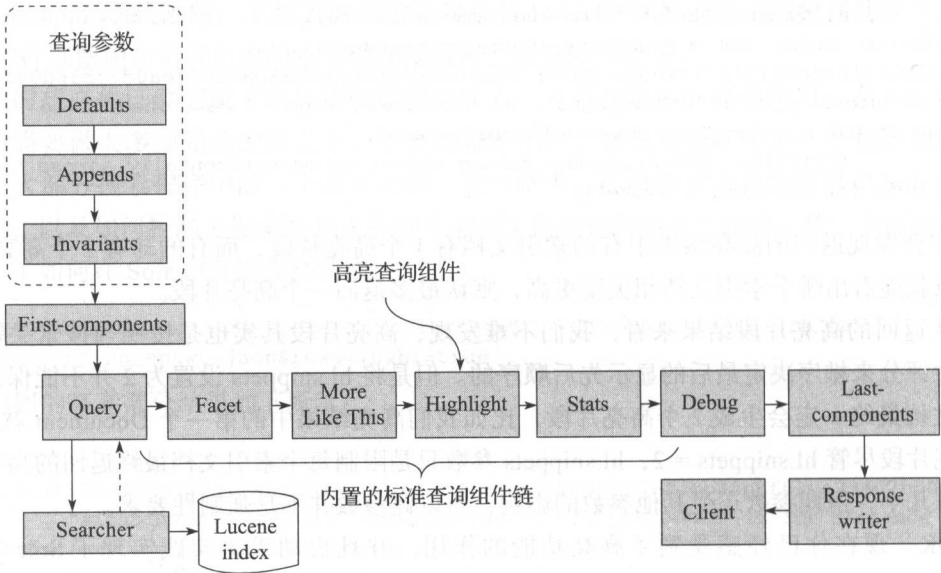


图 7-4 查询组件的工作流程

如图 7-4 中的虚线框内属于查询请求参数初始化过程，主要包括 Defaults、Appends、Invariants。其中 Defaults 表示默认参数，当用户未设置任何查询请求参数时会使用默认参

数, Appends 即用户传递的请求参数, 用于覆盖 Defaults。最后一步 Invariants 用于设置不变参数。请求参数设置完成之后就初始化查询请求组件链, 查询请求组件链总体上又分 3 部分: First、main、last。其中 main 表示主要的查询组件部分, 这里的组件你可以自由定义, 可以是 Solr 内置的那几种查询组件, 也可以是你自定义的查询组件。不同的查询组件按照它们在查询组件链中定义的顺序依次被执行, 这里明显是责任链模式。first 和 last 属于开始和结束组件, 有点类似于 AOP 里的切面概念, 即在查询请求的开始和结束阶段再嵌入两个查询组件, 当你想要在查询开始之前或者在查询结束但查询结果尚未返回给用户之前做些什么, 可能会用到 first-components 和 last-components 查询组件。此外, 从图 7-4 我们还可以得知, 高亮查询组件是 Query 组件的下流, 因此, 高亮组件每次仅仅只是对 Query 组件返回的一页数据进行高亮处理。

假如我们的查询请求参数 rows = 10, 即表示我们每页只返回 10 条结果, 同时也表示高亮组件每次只会处理 10 条结果。也就是说每页返回的结果集数量越大, 那么高亮组件的工作量也就越大, 这就直接影响了高亮组件的查询响应时间。而且, 高亮组件还需要知道你期望对索引文档中的哪个域执行关键字高亮, 正如查询示例中演示的那样, 采用 df 参数指定我们需要在 sighting\_en 域上执行关键字高亮。如果你想要在多个域上执行高亮, 那么你可以通过指定 hl.fl 参数来实现。假如你的索引文档中有 title 和 body 这两个域, 那么这时候你可以指定 hl.fl = title, body。了解了这些之后, 我们需要继续思考一个问题: Solr 高亮组件是如何生成高亮片段的? 答案如图 7-5 所示。

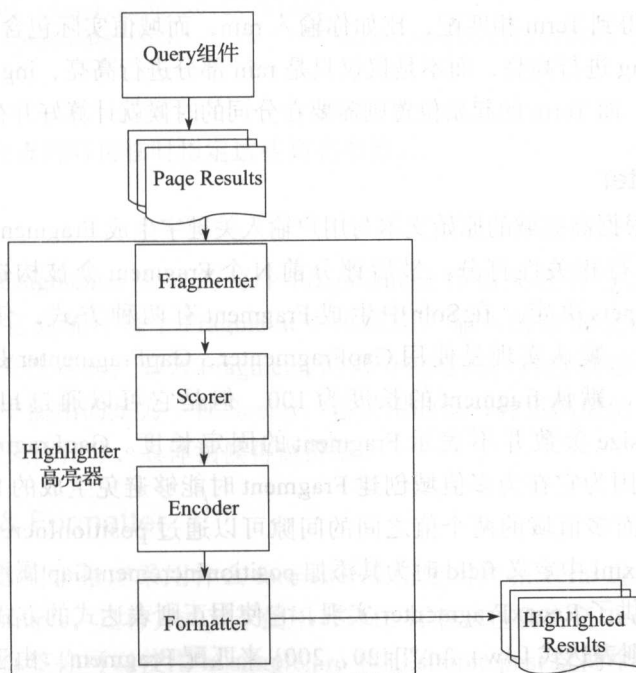


图 7-5 Solr 高亮器工作原理



在 Solr 高亮器执行高亮之前，它需要首先访问域的原始文本即域值，要想访问域的原始文本，那么该域必须 `stored = true`。正如我们的示例中演示的那样，高亮域是 `sighting_en`，它在 `schema.xml` 中是这样定义的：

```
<field name="sighting_en" type="text_en" indexed="true" stored="true" multiValued="true" />
```

获取到高亮域的域值后，高亮器需要根据域类型中配置的索引时使用的分词器对域值进行分词，比如我们的示例中，高亮域是 `sighting_en`，且配置的域类型为 `text_en`，该域类型的配置上面给出。它首先会根据标准分词器 `StandardTokenizerFactory` 对高亮域的域值进行分词，然后依次经过 `StopFilterFactory`、`LowerCaseFilterFactory`、`EnglishPossessiveFilterFactory`、`KeywordMarkerFilterFactory`、`PorterStemFilterFactory` 这几个 `TokenFilter`。假如我们的 `sighting_en` 域的域值是这样的：“It was raining when I saw a blue fireball.”。那么首先会根据空格分出每个单词，然后移除停用词 `It was when I a`。紧接着会进行词干还原：`raining-->rain`。但是高亮器为什么要对其分词呢？主要是出于两方面原因考虑。首先，高亮片段中的 `Term` 需要与用户输入的搜索关键词进行比较。经过分词处理后 `raining` 变成 `rain`，而用户输入的关键词“blue fireball in the rain”中也包含 `rain`。分词后才能与用户输入的关键词相匹配。第二点，当用户输入的查询关键字中包含的 `Term` 与高亮域分词后得到的 `Term` 相匹配了，想要实现高亮，那么对于 Sol 中的 `FastVectorHighlighter` 高亮器需要知道这些 `Term` 在域的域值原始文本中的起始位置，这样才能够在它们两头添加高亮标签。而实际匹配的也是与分词后得到 `Term` 相匹配，比如你输入 `rain`，而域值实际包含的是 `raining`，我们期望是对整个 `raining` 进行高亮，而不是仅仅是 `rain` 部分进行高亮，`ing` 部分不高亮，这不是我们所想看到的。而 `Term` 的起始位置则需要在分词的时候就计算好并存储在索引结构中。

## 7.2.1 Fragmenter

`Fragmenter` 会根据高亮域的原始文本与用户输入关键字生成 `Fragment`，然后交由 `Scorer` 对每个 `Fragment` 进行相关性打分。最后评分前 `N` 个 `Fragment` 会被构造成 `Snippet`，这里的 `N` 由参数 `hl.snippets` 决定。在 Solr 中生成 `Fragment` 有两种方式，使用 `GapFragmenter` 和 `RegexFragmenter`。默认实现是使用 `GapFragmenter`，`GapFragmenter` 是基于一个目标长度来生成 `Fragment`，默认 `fragment` 的长度为 100，但是它可以通过 `hl.fragsize` 参数进行修改。但是 `hl.fragsize` 参数并不表示 `Fragment` 的固定长度。`GapFragmenter` 之所以叫做 `GapFragmenter`，是因为它在为多值域创建 `Fragment` 时能够避免生成的 `Fragment` 跨越了很大一个位置间隙，而多值域的两个值之间的间隙可以通过 `positionIncrementGap` 属性来控制，你可以 `schema.xml` 中定义 `field` 时为其添加 `positionIncrementGap` 属性。

同时 Solr 还提供了 `RegexFragmenter` 实现，它使用正则表达式的方式来生成 `Fragment`。比如你可以使用正则表达式 `[-\w, \/\n"]{20, 200}` 来匹配 `Fragment`。由于 `RegexFragmenter` 并不是默认实现，所以如果你想要默认使用 `RegexFragmenter`，那么你需要在 `solrconfig.xml`



中提前配置 `RegexFragmenter`，配置示例如下所示：

```
<requestHandler name="/select" class="solr.SearchHandler" default="true">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <str name="defType">edismax</str>
    <str name="q.alt">*:*</str>
    <str name="rows">10</str>
    <str name="fl">*,score</str>
    <str name="hl">on</str>
    <str name="hl.snippets">5</str>
    <str name="hl.fragsize">50</str>
    <str name="hl.maxAnalyzedChars">510000</str>
    <str name="hl.requireFieldMatch">true</str>
    <str name="hl.fragmenter">regex</str>
    <str name="hl.fragListBuilder">simple</str>
    <str name="hl.phraseLimit">1000</str>
    <str name="hl.usePhraseHighlighter">true</str>
    <str name="hl.highlightMultiTerm">true</str>
    <str name="hl.useFastVectorHighlighter">true</str>
  </lst>
</requestHandler>
```

```
<fragmenter name="regex" class="solr.highlight.RegexFragmenter">
  <lst name="defaults">
    <int name="hl.fragsize">70</int>
    <float name="hl.regex.slop">0.5</float>
    <str name="hl.regex.pattern">[-\w ,/\n\&quot;&apos;]{20,200}</str>
  </lst>
</fragmenter>
```

当然你也可以在查询时再临时指定这些高亮参数。

## 7.2.2 Scorer

`Scorer` 用于对 `Fragmenter` 生成的每个 `Fragment` 进行相关性评分，默认 `Scorer` 的实现是 `QueryScorer`。它会统计每个 `Fragment` 中出现用户输入的查询关键字的次数，出现次数越频繁说明相关性越高，那么 `Fragment` 的得分就越高，自然就越优先返回。高亮器 `PostingsHighlighter` 中使用的是另一种 `Scorer`，它采用的是一种更高级的打分算法来实现对每个 `Fragment` 的相关性打分，具体稍候再做讲解。

## 7.2.3 Encoder & Formatter

`hl.formatter` 参数用于指定采用什么 `formatter` 来指定对高亮 `Term` 进行格式化，默认实现是 `hl.formatter = simple`，这种方式对于使用 `HTML` 标签来包裹高亮 `Term` 比较适用，设置两头的包裹 `HTML` 标签你可以使用 `hl.simple.pre` 和 `hl.simple.post` 这两个参数，比如你想要使用 `css` 美化高亮 `Term`，那么你在查询请求中如下所示添加 2 个请求参数：

```
hl.simple.pre=<span class="foo">&
hl.simple.post=</span>
```

这样高亮 Term 就被格式化成 `<span class = "foo">term</span>`，之后就可以在你的 CSS 样式文件中任意定义 foo 类样式。

Encoder 组件负责在每个 Fragment 传递给 Formatter 组件之前对特殊字符进行编码，当你想要生成 HTML 标签格式的高亮 Fragment 时，HTML Encoder 会转义 HTML 字符实体，比如双引号 (") 会转义成 `&quot;`。

## 7.3 Facet & Highlighting

OK，现在我们已经掌握了 Solr 在对每个索引文档进行高亮时是如何生成高亮片段的。接下来让我们通过一些示例继续深入学习如何将 Facet 查询与高亮查询结合。

回顾上一章学习的 Facet，它允许用户通过使用 Facet 来改进搜索条件。在我们的 UFO 示例中，有 3 个域适合用于 Facet 查询：shape、location、month。下面的示例演示了在同一个查询中同时使用 Facet 和 Highlighting：

```
http://localhost:8080/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=json&
hl=true&
hl.snippets=2&
hl.fl=sighting_en&
facet=true&
facet.limit=4&
facet.field=shape_s&
facet.field=location_s&
facet.field=month_s
```

在上面的查询中，我们分别在 shape\_s、location\_s 和 month\_s 这 3 个域上执行 Facet 查询，最后 Facet 查询统计的结果如下所示：

```
[Shape]:
fireball(1745)
light (1495)
triangle (781)
circle(761)
```

```
[Location]:
Seattle, WA(88)
Phoenix,AZ(69)
Portland, OR(46)
Houston,TX(44)
```

```
[Month]:
```

July (1145)  
 August (1062)  
 November (924)  
 October (909)

当你单击 Shape 下的 light 分类, 想要了解该分类下的 UFO 事件信息时, 会在之前的 Facet 请求上追加一个 fq 参数即 `fq = shape_s: light`, 在 Facet Query 上应用 Filter Query 我们上一章已经讲解过, 这里的重点是 Highlighting。此时相当于在上一次查询的基础之上查询 `shape = light` 的索引文档, 换句话说, 此时 light 就相当于查询关键字, 那么也就意味着此时高亮器也应该将 Facet 查询返回的 “light” 也进行高亮, 然而, 当我们应用了 Filter Query 之后, Highlighter 高亮器并没有对我们的 “light” 这个 Term 进行高亮, 这是因为 “light” 是在 Filter Query 查询中, 并不存在于 q 参数的主查询中。我们需要做的是如何在不改变查询的基础之上控制高亮器使用的查询 Term, 在 Solr 中你可以通过指定 `hl.q` 参数来实现。下面的示例演示如何使用 `hl.q` 参数将 light 这个 Term 添加高亮器使用的 Query 中:

```
http://localhost:8080/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=json&
hl=true&
hl.snippets=10&
hl.fl=sighting_en&
hl.q=blue fireball in the rain light&
fq=shape_s:light
```

这里我们通过 `hl.q` 参数将 light 这个 Term 加入了高亮匹配 Term 中, 这样只要索引文档中也包含了 light 这个 Term, 那么最终返回的高亮片段中也可能会包含 “light”。

上面的高亮示例中我们的主查询表示匹配索引文档中包含 blue、fireball、rain 其中任意一个 Term。此时如何打开查询的调试模式 (`debug = true`), 你会发现在查询时分词处理完成之后, 执行了:

```
sighting_en:blue OR sighting_en:firebal OR sighting_en:rain
```

这个查询匹配了 9706 个索引文档。假设我期望查询索引文档中包含 “blue fireball” 这个短语, 而不是包含 “blue” 和 “fireball” 这两个 Term, 此时 Solr 应该正确的高亮了 “blue fireball” 短语, 而不高亮单独的 “blue” 和 “fireball”。即如何在高亮查询中使用 Phrase Query 对短语进行高亮, 此时你需要使用 FastVectorHighlighter 高亮器, 关于 FastVectorHighlighter 高亮器的内容请看下一小节。

## 7.4 高亮多值域

Solr 同样支持对多值域进行高亮, 下面的示例演示如何对多值域进行高亮:

```
http://localhost:8080/solr/ufo/select?q=fire cluster clouds thunder&
```

```
df=nearby_objects_en&
wt=json&
hl=true&
hl.snippets=2
```

上面的示例与对单值域进行高亮没什么区别，因为尽管我们在多值域的值的时候，多个值是分开多次 add 的，比如像下面这样：

```
doc.addField("nearby_objects_en", "arc of red fire");
doc.addField("nearby_objects_en", "cluster of dark clouds");
doc.addField("nearby_objects_en", "thunder and lightning");
```

然而实际多个值之间实际是存在间隙（Gap）的，即 positionIncrementGap 的属性值，但 positionIncrementGap 默认值是 0，也就是实际存储的时候你可以认为它们是紧挨在一起的，比如我们的示例中，你可以认为索引文档是存在“clouds thunder”这个 Term 的，所以当你查询“fire cluster clouds thunder”时，包含这 4 个 Term 的高亮结果都返回了。但是默认情况下，Solr 只会返回包含搜索关键字的高亮结果，不包含的不返回。如果你希望返回的高亮结果直接就是该多值域的每个值，只是命中了搜索关键字的会进行高亮，没有命中的就直接显示原始域值，那么此时你需要添加一个 hl.preserveMulti 参数，并将其设置为 true，此参数默认值为 false，它表示在高亮结果中保留多值域的每个值，直接在原始域值的基础上进行高亮，这样便于用户确认是多值域中的哪几个值被高亮了。实际运行效果请大家执行如下高亮查询去感受下：

```
http://localhost:8080/solr/ufo/select?q=fire cluster clouds&
df=nearby_objects_en&wt=json&hl=true&hl.snippets=10&hl.preserveMulti=true
```

## 7.5 高亮参数

在开始学习其他 Solr 高亮器之前，让我们先来总结下 Solr 中默认高亮器支持的请求参数吧。下面的表格展示了 Solr 中的高亮器支持的请求参数，虽然有些参数我们上面并未提及，但大部分参数都是不言自明的，具体请看表 7-2 中的详细解释。

表 7-2 Solr 高亮参数

参数名称	描 述	默认值
hl	表示是否为你的查询开启高亮功能	false
hl.snippets	为每个域生成的高亮片段最大个数	1
hl.fl	为哪些域生成高亮片段，多个域名称之间采用逗号分隔	如果未设置会以 df 参数为准
hl.fragmenter	指定使用什么 Fragmenter 组件来生成 Fragment，Fragmenter 组件需要在 solrconfig.xml 中注册	gap
hl.fragsize	设置每个 fragment 的目标长度，并不是一个严格的字符最大限制	100

(续)

参数名称	描 述	默认值
hl.q	用于高亮的查询, 你可以添加与 q 参数的主查询不同的额外的 Term	没有默认值
hl.alternateField	当没有生成任何高亮片段时, 指定一个存储域 (即 stored = true 的域) 用于显示	没有默认值
hl.formatter	指定使用哪种 Formatter 组件, Formatter 组件需要在 solrconfig.xml 中注册	simple
hl.simple.pre	为每个高亮 Term 的开头添加的高亮标签, 一般是 HTML 中的标签	<em>
hl.simple.post	为每个高亮 Term 的末尾添加的高亮标签, 一般是 HTML 中的标签, 与 hl.simple.pre 搭配使用	</em>
hl.requireFieldMatch	当对多个域进行高亮, 如果此参数设置为 true, 则表示查询结果不为空才会执行高亮, 如果此参数设置为 false, 则表示它可能匹配某个域, 但是却对另一个域进行高亮。如果 hl.fl 参数使用了通配符, 那就表示自动设置此参数为 true。如果你查询的是所有域, 那么还是将此参数设置为 false 吧, 这样便于你清楚到底是哪些域匹配了搜索关键字	false
hl.maxAnalyzedChars	当 Fragmenter 对一个大文本域进行分词时需要设置最大支持对多长的字符进行分词处理。如果不想做任何限制, 那么请设置为 -1	51200
hl.usePhraseHighlighter	如果设置为 true, 则表示高亮器只对匹配的短语进行高亮, 匹配的单个 Term 不进行高亮, 且 Solr 会使用 Lucene 中的 SpanScorer 去对 Phrase 进行打分	false
hl.mergeContiguous	若此参数设置为 true, 那么 Solr 会将相邻的 Fragment 合并为一个 Fragment	false (为了向后兼容)
hl.highlightMultiTerm	若此参数设置为 true, 即表示启用高亮器对 range/wildcard/fuzzy/prefix 这些查询的支持	false
hl.preserveMulti	若此参数设置为 true, 则表示对多值域的每个值执行高亮处理, 不管该值是否与搜索关键字匹配。并且返回的高亮结果会保留多值域的原始域值的添加顺序	false
hl.maxMultiValuedToExamine	限制最多对多少个多值域的域值进行检查	Integer.MAX_VALUE
hl.maxMultiValuedToMatch	限制最后有多少个多值域的域值匹配, 当 hl.maxMultiValuedToExamine 参数也设置了, 那么哪个参数先达到限制就终止	Integer.MAX_VALUE
hl.maxAlternateFieldLength	设置 hl.alternateField 参数指定的域的字符最大长度, 设置为小于等于零的数值表示不做限制	无限制
hl.highlightAlternate	如果此参数设置为 true 且设置了 hl.alternateField, Solr 会显示整个 alternate field 并显示高亮, 如果 hl.maxAlternateFieldLength = N 参数设置了, 那么 Solr 会返回最多 N 个字符作为高亮摘要。如果此参数设置为 false, 或者 hl.alternateField 参数指定的可选域没有匹配搜索关键字, 那么会直接显示可选域的域值文本但不包含高亮片段	true

(续)

参数名称	描 述	默认值
hl.tag.pre hl.tag.post	与 hl.simple.pre、hl.simple.post 参数类似，用于 Postings-Highlighter 高亮组件	<em></em>
hl.phraseLimit	用于提升 FastVectorHighlighter 高亮器的执行性能，表示最多对多少短语进行匹配	Integer.MAX_VALUE
hl.fragListBuilder	用于指定使用什么类型 SolrFragListBuilder。其他可选值有 single、simple，single 它会将整个域值当作一个高亮片段	weighted
hl.fragmentsBuilder	fragments builder 主要负责格式化 Fragment，它默认会 <em> 和 </em> 装饰高亮 Term。用于指定使用什么类型 SolrFragmentsBuilder。 SolrFragmentsBuilder 配置示例请接着往下看	default
hl.regex.slop	表示 hl.fragsize 会发生变化以适应正则表达式的因子，默认值是 0.6，意思是如果 hl.fragsize = 100 那么 fragment 的大小会从 40 ~ 160。当你使用 Regex-Fragmenter 时此参数会有用	0.6
hl.regex.pattern	当你使用 RegexFragmenter 时需要用到的正则表达式，根据指定的正则表达式来匹配	无默认值
hl.regex.maxAnalyzedChars	当你使用 RegexFragmenter 时需要用到此参数，用于限制 RegexFragmenter 只处理限定字符长度范围内的域值进行处理	10000
hl.boundaryScanner	用于配置如何确定 Fragment 的边界，默认是以字符级别来划分边界，可选值有 breakIterator、simple。SimpleBoundaryScanner 会根据 hl.fragsize 参数决定的关键字的起始偏移量和结束偏移量，重新计算摘要的起始偏移量。关于如何在 solrconfig.xml 中配置 Boundary-Scanner 请接着往下看	simple
hl.bs.maxScan	用于指定 SimpleBoundaryScanner 边界扫描器扫描字符的长度	10
hl.bs.chars	用于指定能够确定 Fragment 边界的字符	.,!/?\t\n 以及空格
hl.bs.type	决定 BreakIterator 怎么划分界定符，可选值有：CHARACTER、WORD、SENTENCE、LINE、WHOLE。SENTENCE 是按句子来划分	WORD
hl.bs.language	当你使用 BreakIteratorBoundaryScanner 时会需要此参数，用于指定 Local 使用什么本地语言	空字符串
hl.bs.country	当你使用 BreakIteratorBoundaryScanner 时会需要此参数，用于为 Local 类指定国家信息	空字符串

上面表格中的参数你都可以单独每个域进行设置，举个例子，假如你有 title 和 body 这两个域，title 通常是一些比较短的域，所以一般一个高亮片段就够了，但是你可能希望为 body 域生成 3 个高亮片段，那么可以这样处理：



```
f.body.hl.snippets=3
```

通常，你可以在域级别去覆盖全局的高亮参数，设置语法为：`f.<fieldname>.<highlight-param>`。下面是 `solrconfig.xml` 中 `fragmentsBuilder` 的配置示例：

```
<fragmentsBuilder name="colored" class="org.apache.solr.highlight.ScoreOrder-
FragmentsBuilder">
  <lst name="defaults">
    <str name="hl.tag.pre"><![CDATA[
    <b style="background:yellow">,<b style="background:lawgreen">,
    <b style="background:aquamarine">,<b style="background:magenta">,
    <b style="background:palegreen">,<b style="background:coral">,
    <b style="background:wheat">,<b style="background:khaki">,
    <b style="background:lime">,<b style="background:deepskyblue">]]></str>
    <str name="hl.tag.post"><![CDATA[</b>]]></str>
  </lst>
</fragmentsBuilder>
```

下面是 `solrconfig.xml` 中 `boundaryScanner` 的配置示例：

```
<boundaryScanner name="breakIterator" class="solr.highlight.BreakIterator-
BoundaryScanner">
  <lst name="defaults">
    <str name="hl.bs.type">WORD</str>
    <str name="hl.bs.language">en</str>
    <str name="hl.bs.country">US</str>
  </lst>
</boundaryScanner>
```

## 7.6 FastVectorHighlighter

使用默认的高亮器最大的问题就是对于大文本域执行高亮查询会非常慢。导致查询速度慢的主要原因就是它需要在查询时对域值文本进行重新分词。为了解决这个问题，Solr 提供了 `FastVectorHighlighter` 快速高亮器，它的执行速度比默认高亮器要快，因为它跳过了在生成 `Fragment` 阶段需要重新分词的步骤。



**注意** 默认高亮器性能问题并不是很突出，在我们的 UFO 示例中甚至很难重现这种问题，默认高亮器通常运行速度很快，甚至当你将每页返回的索引文档大小设置为 50，性能问题依然不是很突出。但是如果你每页返回的索引文档成千上万了，又或者需要同时对多个域进行高亮，那么此时默认高亮器的性能就显现出来了，变得越来越慢，尽管当默认高亮器运行速度很慢时我们可以使用 `FastVectorHighlighter` 高亮器来救急，但是并不意味着默认高亮器就 100% 不适合于你的应用，也并不意味着 `FastVectorHighlighter` 高亮器在任何时候都比默认高亮器优秀。

FastVectorHighlighter 高亮器在高亮时需要访问每个 Term 的位置信息以及偏移量信息，所以它依赖于你在索引创建时就提前计算好并存储到索引结构中。因此，使用 FastVectorHighlighter 高亮器时，任何需要高亮的域都必须在索引创建时启用 termVectors、termPositions、termOffsets。在我们的示例中，如果我们想要为 sighting\_en 域启用 FastVectorHighlighter 高亮器，那么需要如下进行定义：

```
<field name="sighting_en" type="text_en" indexed="true" stored="true"
termVectors="true" termPositions="true" termOffsets="true"/>
```

这看起来似乎很简单，但是修改完域的定义之后，别忘了，你需要对所有索引文档重新建立索引，这样才能使你的设置立即生效。但是带来更大的问题是这些属性会使得你的索引体积变得臃肿，对于 UFO 示例来说，开启那 3 个属性重建索引之后，索引体积由原来的 69MB 增加到 109MB，这种索引体积的增长对于小数据集来说可能微不足道，但是对于大规模的索引文档来说，那简直就是灾难。存储 vectors、positions、offsets 这些信息同时还会稍微减缓你索引创建的速度，在高吞吐量环境下要求能够近实时搜索时这可能会是一个问题。

想要使用 FastVectorHighlighter 高亮器，你需要首先在 schema.xml 中修改域的定义启用那 3 个属性，然后重新启动 Solr Server 服务，紧接着你需要对所有索引文档重新创建索引。想要激活 FastVectorHighlighter 高亮器，你需要传递 hl.useFastVectorHighlighter 参数并将其设置为 true，hl.useFastVectorHighlighter 参数使用示例如下所示：

```
http://localhost:8080/solr/ufo/select?q="blue fireball"&
df=sighting_en&
wt=json&
hl=true&
hl.snippets=10&
hl.useFastVectorHighlighter=true
```

在上面的查询示例中，我们的高亮返回结果会将整个查询短语进行高亮，而不仅仅只是对单个高亮 Term 进行高亮，这也是 FastVectorHighlighter 高亮器与默认高亮器相比，它的另一个优势。

## 7.7 PostingsHighlighter

Solr 中还提供了一种新的高亮器：PostingsHighlighter，它的执行速度比 FastVectorHighlighter 还快。PostingsHighlighter 需要在倒排索引表中存储 Term 的偏移量，与 FastVectorHighlighter 相比，FastVectorHighlighter 需要在索引中创建一个独立的数据结构用于检索 Term 的位置信息和偏移量。回顾下 Lucene 的倒排索引表，我们知道在 Lucene 的倒排索引表中的倒排索引信息是存储在 posting list 中的，其中包含 term、term 出现的 document 列表、term frequency。如果想要使用 PostingsHighlighter，你还需要在创建索引时在倒排索引

表的 posting list 中再额外存储 Term 的位置信息和偏移量信息，在 Solr 中，你需要做的就是必须在 schema.xml 中为高亮域添加 storeOffsetsWithPositions 属性，并将其设置为 true，配置示例如下所示：

```
<field name="sighting_en" type="text_en" indexed="true" stored="true"
storeOffsetsWithPositions="true"/>
```

同理，当你修改了 schema.xml 中域的定义信息，那么需要重新建立你的索引，然后重启你的 Solr Server 服务或者重新加载 Core。启用 storeOffsetsWithPositions 之后，索引体积只是从 69MB 增大到 85M，相比 FastVectorHighlighter 而言，索引体积增长的并不是那么迅猛了，这对于大规模的索引来说具有重大意义。

想要使用 PostingsHighlighter 高亮组件，你还需要在 solrconfig.xml 中显式的注册它，下面是配置示例：

```
<requestHandler name="standard" class="solr.StandardRequestHandler">
<lst name="defaults">
<int name="hl.snippets">10</int>
<str name="hl.tag.pre">&lt;em&gt;</str>
<str name="hl.tag.post">&lt;/em&gt;</str>
<str name="hl.tag.ellipsis">... </str>
<bool name="hl.defaultSummary">true</bool>
<str name="hl.encoder">simple</str>
<float name="hl.score.k1">1.2</float>
<float name="hl.score.b">0.75</float>
<float name="hl.score.pivot">87</float>
<str name="hl.bs.language"></str>
<str name="hl.bs.country"></str>
<str name="hl.bs.variant"></str>
<str name="hl.bs.type">SENTENCE</str>
<int name="hl.maxAnalyzedChars">10000</int>
</lst>
</requestHandler>

<searchComponent class="solr.HighlightComponent" name="highlight">
<highlighting class="org.apache.solr.highlight.PostingsSolrHighlighter"/>
</searchComponent>
```

由于 PostingsSolrHighlighter 高亮器在 Solr 5.3.1 版本中还只是实验性功能，并没有提供类似 FastVectorHighlighter 高亮器的请求参数 hl.useFastVectorHighlighter。具体 Solr 支持哪些高亮请求参数，请查阅 org.apache.solr.common.params.HighlightParams 类，因为每个版本经过迭代后支持的功能特性会有所不同。

PostingsSolrHighlighter 高亮器与其他高亮器不同的是，多个高亮片段是采用“...”省略号的形式拼接在一起形成一个高亮片段返回，你可以通过 hl.tag.ellipsis 参数来修改默认的拼接符。

除了高亮执行速度块以及索引创建开销减少了之外，PostingsHighlighter 高亮器还使用了以各种更高级的相似度计算，被称作“BM25”，用于对 Fragment 进行相关性打分，默认的 Scorer 打分器是统计查询 Term 在每个 Fragment 中的出现频率，而 BM25 是一种先进的 tf-idf 打分函数，用于计算索引文档或 Fragment 与 Query 之间的相似性。BM25 Scorer 默认会提高包含在索引中出现不那么频繁的 Term 的 Fragment 的权重。

PostingsHighlighter 高亮器的主要缺点就是它需要 Term 精确的位置偏移量信息，这些信息需要在索引创建时提前设置。因此在使用 PostingsHighlighter 高亮器之前，你需要对高亮域所应用的域类型配置的分词器进行测试，确保分词器正确处理了每个 Token 的位置偏移量信息，即 token 的 start、end、position 数据，你可以通过 Solr Web 后台左侧的 Analysis 菜单提供的分词器测试功能辅助你进行分词测试。

PostingsHighlighter 高亮器支持的配置参数如下所示：

- ❑ hl.q (string) 用于高亮的查询，你可以在这里添加额外的 Term；
- ❑ hl.fl (string) 指定高亮域；
- ❑ hl.snippets (int) 指定返回的高亮片段最大个数，并不是严格固定的限制；
- ❑ hl.tag.pre (string) 指定在高亮 Term 之前添加的文本，一般为 HTML 标签；
- ❑ hl.tag.post (string) 指定在高亮 Term 之后添加的文本，一般与 hl.tag.pre 参数搭配使用；
- ❑ hl.tag.ellipsis (string) 用于指定连接多个高亮段的拼接符，默认为…省略号；
- ❑ hl.defaultSummary (bool) 指定计算查询 Term 与 Fragment 之间的相关度；
- ❑ hl.encoder (string) 高亮片段编码器，可选值有 html 和 simple；
- ❑ hl.score.k1 (float) 指定 bm25 打分参数 'k1'；
- ❑ hl.score.b (float) 指定 bm25 打分参数 'b'；
- ❑ hl.score.pivot (float) 指定 bm25 打分参数 'avgdl'；
- ❑ hl.bs.type (string) 指定如何将文本分成 Fragment，可选值有：[SENTENCE, LINE, WORD, CHAR, WHOLE]；
- ❑ hl.bs.language (string) 指定 BreakIterator 需要使用的本地语言编码，默认值空字符串；
- ❑ hl.bs.country (string) 指定 BreakIterator 需要使用的国家编码，默认值空字符串；
- ❑ hl.bs.variant (string) 指定 BreakIterator 需要使用的变量；
- ❑ hl.maxAnalyzedChars 指定索引文档中域值最大字符长度，超过不处理；
- ❑ hl.multiValuedSeparatorChar 指定多值域的多个值之间的逻辑分隔符；
- ❑ hl.highlightMultiTerm 为 range/wildcard/fuzzy/prefix queries 这一类查询启用高亮功能。

## 7.8 本章总结

在本章中，我们通过一个简单示例讲解了高亮器在 Solr 的使用，详细讲解了 Solr 中高亮器的工作原理，并详细解释了高亮器支持的请求参数，然后介绍了如何使用 FastVector-

Highlighter 和 PostingsHighlighter 这两种高级高亮器，以及如何使用它们来提升高亮查询的性能。

默认的高亮器简单易用，但对于大文本域或者大规模的索引文档，可能查询会比较慢，在这种情况下你可以考虑使用 FastVectorHighlighter，FastVectorHighlighter 高亮器带来了高亮查询的性能提升的同时也是以增大索引体积为代价的。新引入的 PostingsHighlighter 高亮器，它比 FastVectorHighlighter 高亮器性能更高，而且索引体积增大幅度也没 FastVectorHighlighter 高亮器大，因此建议使用 PostingsHighlighter 高亮器，尤其是当你需要对大文本域进行高亮时。

PostingsHighlighter 高亮器最大的不足就是它依赖于 Term 精确的位置信息和偏移量信息，然而并不是所有的分词器都正确处理了 Term 的位置信息和偏移量信息，因为 PostingsHighlighter 高亮器可能并不兼容所有分词器，比如 WordDelimiterFilter。在下一章中，我们将继续学习 Solr 中另一个核心功能：Suggestion。

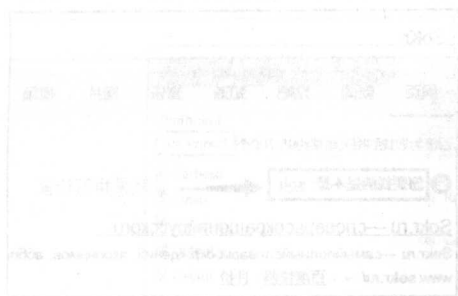


图 2-1-2 高亮器配置界面

## Solr Query Suggestion 查询建议

通过第 8 章，你将可以学习到如下内容：

- 掌握 Solr 中的 Spell-Check 查询组件基本用法；
- 掌握 Solr 中的 Autosuggest 查询组件基本用法；
- 学会基于 N\_Gram 实现 Autosuggest 查询；
- 学会基于用户的过去行为来实现 Autosuggest 查询。

在本章中，我们将介绍两个简单功能来提升你的搜索程序的易用性：Spell-Checking（拼写检查）和 Autosuggest（自动建议）。Spell-Checking（拼写检查）会定位到拼写错误的查询关键字，而拼写错误的查询关键字可能会导致查询没有返回结果。当你使用搜索引擎的时候或多或少都使用过它，比如当你搜索“soKr”，如图 8-1 所示，搜索引擎会提示“您是不是要找 solr”，这就是 Spell-Checking（拼写检查）功能。

Spell-Check 功能至关重要，它能帮助用户不需要花费太多时间去提交一个完美无误的查询，尤其是在收集移动设备上。

Autosuggest 能在用户搜索的时候即时的给予用户一些建议提示，从而帮助用户能够快速构建自己的查询。下面我们开始学习 Solr 中的 Spell-Check 查询组件，以及以 Spell-Check 查询组件为基石的 Autosuggest 功能，如图 8-2 所示的 Autosuggest 功能示例。

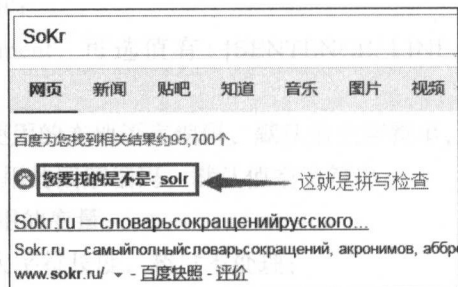


图 8-1 百度中的拼写检查功能



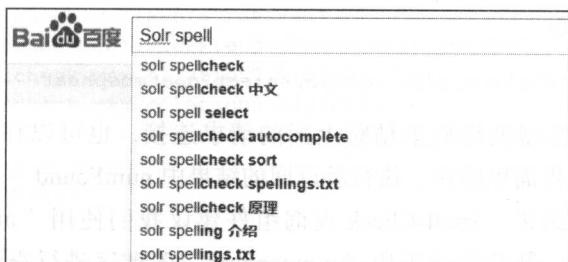


图 8-2 百度中的 Autosuggest 功能示例

## 8.1 Spell-Check

在本章中，你将学习到如何使用的 Solr 的 Spell-Check 查询组件。自动化的 Spell-Check 拼写检查是 Solr 中的核心查询功能，大部分用户期望在查询时无需过多思考如何拼写正确就能正常查询。

### 8.1.1 Spell-Check 简单示例

开始学习 Spell-Check 之前，我们首先需要搭建一个 Spell-Check 示例，这里为了简便，就不一一介绍搭建过程了，读者请自行从我的 Github 上下载随书源码，在 `example-docs\ch08\cores` 目录下可以获取到我们为本章学习搭建的 "solrpedia" Core，请将其复制到 SOLR\_HOME 目录下，然后重启 Solr Server 服务，Solr 会自动发现我们的 "solrpedia" Core，然后你可以通过 Solr Web 后台提供的 DataImport 功能界面导入我们的测试数据，导入成功后会有 13000 条索引数据，如图 8-3 所示。

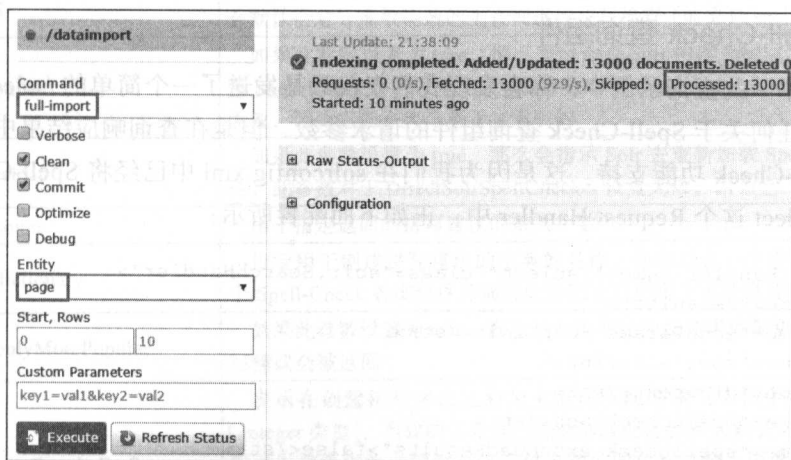


图 8-3 将 solrpedia.xml 导入到 "solrpedia" Core 中

我们执行如下查询：

```
http://localhost:8080/solr/solrpedia/select?q=atmosphear
```

你可以直接在浏览器地址栏里粘贴上面的请求连接，也可以在 Solr Web 后台左侧的 Query 菜单点开的功能界面里操作。执行后返回的结果中 numFound = 0 表示根据 atmosphear 关键字没查询到任何结果，Spell-Check 查询组件建议我们使用 “atmosphere” 关键字进行查询，然后 collations 部分提示采用 “atmosphere” 关键字进行查询，命中了 86 个索引文档。

你可以根据 collation 部分提供的信息自动去根据 Spell-Check 建议的查询关键字去查询，同时可以提示用户你是不是要找 “atmosphere”。但是假如你执行下面这个查询：

```
http://localhost:8080/solr/solrpedia/select?q=title:anaconda
```

你会发现原始查询没有返回结果，同时也没有任何查询建议返回。这个查询示例演示了用户查询的关键字在索引中不存在的情况。在这种情况下，既没有匹配任何索引文档，Solr 也无法提供任何建议。当遇到这种情况，你需要做的就是将哪些不存在于索引中的查询关键字添加到外部字典中。

下面这个查询示例演示了当用户拼写正确，但 Solr 仍然提供了拼写建议：

```
http://localhost:8080/solr/solrpedia/select?q=Julius&df=suggest&fl=title&wt=json&indent=true
```

从返回的查询结果中我们可以得知，用户提供的查询关键字匹配了 4 个索引文档，而 Solr Spell-Check 提供的两个拼写建议匹配的索引文档数量都小于用户提供的查询关键字匹配的索引文档数量，因此这个时候你不需要提供拼写建议提示。

### 8.1.2 Spell-Check 查询组件

在上一章节的查询示例中，你会发现我们仅仅只是发送了一个简单的 /select 查询请求，并没有传递任何关于 Spell-Check 查询组件的请求参数，但是在查询响应结果中却已经自动获取了 Spell-Check 功能支持。这是因为我们在 solrconfig.xml 中已经将 Spell-Check 查询组件集成到 /select 这个 Request Handler 中，正如下面配置所示：

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="df">text</str>
    <str name="spellcheck">on</str>
    <str name="spellcheck.extendedResults">false</str>
    <str name="spellcheck.count">5</str>
    <str name="spellcheck.alternativeTermCount">2</str>
    <str name="spellcheck.maxResultsForSuggest">5</str>
```

```

<str name="spellcheck.collate">true</str>
<str name="spellcheck.collateExtendedResults">true</str>
<str name="spellcheck.maxCollationTries">5</str>
<str name="spellcheck.maxCollations">3</str>
</lst>
<arr name="last-components">
<str>spellcheck</str>
</arr>
</requestHandler>

```

通过上面这种方式，相当于全局性地为所有搜索查询都启用了 Spell-Check 功能支持，此外，这样也避免了搜索客户端在每个查询请求中需要去传递大量的 Spell-Check 请求参数。如果你的某个查询请求不希望启用 Spell-Check 功能，那么可以在该请求中传递 `spellcheck = false` 参数即可实现禁用。同时，将 Spell-Check 组件作为查询请求链条中的最后一个环节也是非常有意义的，因为我们仍然希望那些默认的查询组件比如 `query`、`facet`、`debug` 等也能在查询处理过程中被执行。如果你通过传递 `spellcheck.collate = true` 参数启用了 Spell-Check 中的 `collation`，正如我们在配置文件中配置的那样，那么 Spell-Check 查询组件必须作为查询请求处理链条中的最后一个组件，因为生成 `collation` 查询需要 Query 查询组件已经被执行，通常情况下，建议将 Spell-Check 配置为查询请求处理链条当中的最后一个查询组件。

表 8-1 所示列出了你可以为 Spell-Check 查询组件指定的可用请求参数及其详细描述。

表 8-1 Spell-Check 查询组件的可用参数

参数名称	描 述
<code>spellcheck</code>	表示是否为你的查询请求启用 Spell-Check 查询组件，默认值 <code>false</code> 即不启用
<code>spellcheck.q</code>	用于指定 <code>spell-check</code> 对哪个查询进行拼写检查，如果此参数未指定，那么默认会对 <code>q</code> 参数构造的主查询进行拼写检查（前提是 <code>spellcheck = true</code> ）
<code>spellcheck.build</code>	如果此参数设置为 <code>true</code> ，那么表示命令 Solr 构建 Spell-Check 字典（前提是 Spell-Check 字典不存在），此参数对于 <code>DirectSolrSpellChecker</code> 来说设置无效，因为它的字典是基于主索引的
<code>spellcheck.reload</code>	如果此参数设置为 <code>true</code> ，那么会指示 Solr 去重新加载 Spell-Check 查询组件，此参数对于 <code>DirectSolrSpellChecker</code> 设置无效，因为它一直都是最新的
<code>spellcheck.count</code>	用于指定返回的拼写建议的最大个数
<code>spellcheck.dictionary</code>	指定用于创建拼写建议的字典的名称，你可以在一个查询请求中激活多个 Spell-Check 查询组件并通过此参数为它们每个单独设置字典
<code>spellcheck.onlyMorePopular</code>	如果此参数设置为 <code>true</code> ，那么只有那些比原始请求匹配更多索引文档的拼写建议会被返回
<code>spellcheck.maxResultsForSuggest</code>	表示在创建拼写建议之前为主查询匹配的索引文档个数设置一个阈值（Integer 类型），当你的主查询已经能够匹配足够多的索引文档时，设置此参数可以避免你为创建拼写建议付出额外的执行开销，因此此时已经没必要提供拼写建议了。比如当你将 <code>spellcheck.maxResultsForSuggest</code> 参数设置为 10，而你的主查询匹配了 20 个索引文档，那么此时拼写建议将不会被启用

参数名称	描 述
spellcheck.accuracy	此参数为 float 类型，取值范围为 [0, 1] 之间，如果拼写建议是合法的，此参数值越大，则表示返回的拼写建议越精确，但同时也会导致返回的拼写建议个数越少。
spellcheck.extendedResults	表示是否返回关于拼写建议更多信息，比如建议的 Term 在索引文档中的出现频率
spellcheck.collate	如果此参数设置为 true，那么 Solr 会基于拼写建议推荐的一系列 Term 去构建额外的查询，并返回每个 Term 匹配的索引文档的个数。当你的用户单击了 "你是不是要找 xxx" 链接之后，你的搜索客户端可以执行 collate 查询。collate 查询会保证返回一些索引文档，而这意味着 Solr 必须在后台隐式的执行 collate 查询
spellcheck.maxCollations	用于限制 Solr 生成的 collation 查询的最大个数 (Integer 类型)
spellcheck.maxCollationTries	指定 collation 最大尝试次数，此参数值越小，执行性能越好

我们在 SearchHandler 中配置了 spellchecker 为我们查询请求链条中的最后一个查询组件，而 spellcheer 查询组件也需要在 solrconfig.xml 中注册，配置示例如下所示：

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <str name="queryAnalyzerFieldType">text_suggest</str>
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">suggest</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    <str name="distanceMeasure">internal</str>
    <float name="accuracy">0.5</float>
  </lst>
  <lst name="spellchecker">
    <str name="name">wordbreak</str>
    <str name="classname">solr.WordBreakSolrSpellChecker</str>
    <str name="field">suggest</str>
    <str name="combineWords">true</str>
    <str name="breakWords">true</str>
    <int name="maxChanges">10</int>
    <int name="minBreakLength">5</int>
  </lst>
</searchComponent>
```

DirectSolrSpellChecker 是 Solr 中的 Spell-Checker 查询组件默认实现。这个查询组件是直接基于主索引来提供拼写建议的。在之前的 Solr 版本中，你需要为 Spell-Check 基于主索引库单独创建一个索引库，采用两个索引库的话你需要维护两个索引库，同时当主索引库数据更新了你的 Spell-Check 使用的索引库也需要随之更新，这简直就是噩梦。DirectSolrSpellChecker 查询组件提供了很多参数供你去调整它的内部行为，有 3 个比较重要的参数需要引起你的注意：field、distanceMeasure、accuracy。field 参数表示你需要在哪个域上执行拼写检查并提供拼写建议。在示例中，我们设置的是 suggest 域，suggest 域在 schema.xml

中定义如下：

```
<field name="suggest" type="text_suggest" indexed="true" stored="false" />
```

suggest 域的域值是通过 <copyField> 复制域从 title 域上复制过来的：

```
<copyField source="title" dest="suggest"/>
```

使用复制域允许你对 title 域的域值文本应用不同的分词器进行分词处理。

distanceMeasure 参数用于指示 Solr 如何去确定拼写建议。Spell-Checker 会使用一些函数来计算查询 Term 和字典中每个 Term 的编辑距，正如 Fuzzy Query 实现原理一样。所谓编辑距其实你可以这么去理解：就是一个 Term 变成另一个 Term 需要添加或删除或更新几次，每次只能更改一个字符。比如 atmosphear 和 atmosphere 之间的编辑距是 2，先删除 atmosphear 倒数第二个字符 a，然后在末尾插入 e，所以编辑距是 2。用于计算两个 Term 之间的编辑距有个著名的算法就是 Levenshtein distance。

accuracy 参数决定了返回的拼写建议的精确度，它是一个 [0, 1] 之间的 Float 类型的浮点数。accuracy 参数值越大，表示返回的拼写建议精度越高，但是返回的拼写建议个数会越少。如果 accuracy 参数值设置得太小，Solr 会生成更多的拼写建议但是它们可能精度不高对用户来说没有太大意义。

接下来让我们来看看 queryAnalyzerFieldType 这个配置，它表示使用什么域类型来对查询 Term 进行分词处理以及生成拼写建议。

当你对查询 Term 执行分词处理时，大多数情况下，你希望分词器处理能够尽量最小化，特别是尽量避免使用 tokenFilter 使得 Term 发生剧烈变化。正如下面配置示例所示，我们使用的是 text\_suggest 这个域类型：

```
<fieldType name="text_suggest" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
  </analyzer>
</fieldType>
```

回顾下我们配置的 DirectSolrSpellChecker，它使用的是 suggest 域来构建 Spell-Check 字典，而 suggest 域使用的域类型就是 text\_suggest。通常情况下，你应该使用与构建 Spell-Check 字典一样的分词器来对查询 Term 进行分词处理，这样保证了 schema 在查询 Term 与字典中的 Term 之间能够兼容。

Solr Spell-Check 中的一个强大功能就是你可以将多个 Spell-Check 查询组件结合在一起

构建成一个链条形成一个组合式 Spell-Check。例如，你可以在 <searchComponent> 元素中配置多个 Spell-Check 组件，我们的示例中，还配置了一个 WordBreakSolrSpellChecker 组件，它通过将查询 Term 分割成多个部分或者结合查询 Term 来生成拼写建议。考虑这样一个拼写错误的查询：northatlantic curent，用户打算查找跟“northatlantic curent”相关的话题，默认的 Spell-Checker 会很轻松的为“curent”返回一个拼写建议，但是却没有返回对“northatlantic”的拼写建议。正如下面的查询示例所示：

```
http://localhost:8080/solr/solrpedia/select?
q=northatlantic curent&df=suggest&wt=json&indent=true
```

我们需要的是 Solr 能够识别到“northatlantic”其实是两个 Term，因为我们的用户在输入的时候并没有使用空格将它们分割开。通常，在两个查询 Term 之间忘记输入空格分割是一种比较常见的用户错误。因此我们需要替用户纠正它并提供拼写建议。对于这种情况，WordBreakSolrSpellchecker 能够将查询 Term 分割成多个 Term 然后提供正确的拼写建议。下面的查询示例演示了 WordBreakSolrSpellchecker 如何使用：

```
http://localhost:8080/solr/solrpedia/select?
q=northatlantic curent&
wt=json&indent=true&df=suggest&q.op=AND&
spellcheck.dictionary=wordbreak&
spellcheck.dictionary=default
```

上面的这个查询示例演示了如何将默认的 Spell-Checker 和 wordbreakspell-checker 在同一个查询请求中结合到一起。这样你的用户就能够输入“northatlantic”查询关键字，而 Solr 后台会自动去执行“north atlantic”查询。这里的关键点就是 Solr 提供了一种非常强的灵活性，允许你结合多个 Spell-Chcker 组件来生成拼写建议。

OK，目前你应该已经熟悉了 Solr 中的 Spell-Check 组件，让我们进入下一项学习：Autosuggest。

## 8.2 Autosuggest

Solr 的 Spell-Check 是一个不错的功能，但是你也可以尽力在第一时间根据用户输入的查询 Term 给予拼写建议，从而避免用户出现拼写错误。Autosuggest 功能在手机等移动设备下尤其有用。在本章中，我们将关注 Solr 中的 Autosuggest 功能。在我们开始了解 Solr 中的 Autosuggest 是如何工作的之前，我们先思考下 Autosuggest 功能需要满足哪些硬性指标：

- 首先，它的执行速度必须要快，Autosuggest 必须在用户每输入一个字符之后就能立即更新拼写建议；
- 然后它必须按照 term 的出现频率按序排列，出现频率高的应该靠前显示。

基于以上对 Autosuggest 基本概念的理解，让我们看看它在 Solr 中是如何使用的吧！首先你需要在 solrconfig.xml 中注册 Autosuggest 组件，然后在 request handler 中启用 Autosuggest



组件。下面的配置示例演示了如何在查询请求中启用 Autosuggest 组件：

```
<requestHandler name="/suggest"
class="org.apache.solr.handler.component.SearchHandler">
<lst name="defaults">
<str name="echoParams">none</str>
<str name="wt">json</str>
<str name="indent">true</str>
<str name="spellcheck">true</str>
<str name="spellcheck.dictionary">suggestDictionary</str>
<str name="spellcheck.onlyMorePopular">true</str>
<str name="spellcheck.count">5</str>
<str name="spellcheck.collate">>false</str>
</lst>
<arr name="components">
<str>suggest</str>
</arr>
</requestHandler>
```

在上面的配置示例中我们重新定义了一个 Request Handler，且定义了 Request Handler 包含的查询组件只有 suggest 这一个，它覆盖了默认的组件栈（比如 query、facet、highlighting、debug 等），因为 Autosuggest 要求就是尽可能快的返回拼写建议给用户，因此我们不需要额外执行其他查找组件。这里通过将查询 URL：/suggest 与 SearchHandler（实际查询请求处理类）进行映射，这样查询客户端需要发送 /suggest 请求来获取拼写建议。下面的查询示例演示了如何发起了一个 /suggest 查询请求：

在上面的配置示例中我们重新定义了一个 Request Handler，且定义了 Request Handler 包含的查询组件只有 suggest 这一个，它覆盖了默认的组件栈（比如 query、facet、highlighting、debug 等），因为 Autosuggest 要求就是尽可能快的返回拼写建议给用户，因此我们不需要额外执行其他查找组件。这里通过将查询 URL：/suggest 与 SearchHandler（实际查询请求处理类）进行映射，这样查询客户端需要发送 /suggest 请求来获取拼写建议。下面的查询示例演示了如何发起了一个 /suggest 查询请求：

```
http://localhost:8080/solr/solrpedia/suggest?q=atm&wt=json&indent=true
```

查询返回的结果如下所示：

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 32,
    "spellcheck": {
      "suggestions": [
        "atm", {
          "numFound": 2,
          "startOffset": 0,
          "endOffset": 3,

```

```
"suggestion": [{"atmosphere",
                  "atmospheric"}] ] ] }
```

从上面返回的查询结果中我们可以得知，Autosuggest 组件根据用户输入的“atm”返回了两个拼写建议：“atmosphere”和“atmospheric”。在脑海中想象一下，这个过程就相当于你在搜索输入框中输入 atm 3 个字符之后，Autosuggest 组件自动返回了 2 个拼写建议，建议你采用“atmosphere”或“atmospheric”作为查询关键字进行搜索，这就是 Autosuggest。由于 Autosuggest 组件返回的拼写建议一般肯定是拼写正确的，用户如何选择采用拼写建议的其中一项，也就意味着在用户查询之前，Autosuggest 就提前避免拼写错误的发生。

当/suggest 请求 URL 映射的 Request Handler 接收一个查询请求时，它执行 suggest 组件来生成拼写建议，而 suggest 组件其实是需要提前在 solrconfig.xml 配置文件中注册的，下面是 Autosuggest 组件的注册配置示例：

```
<searchComponent class="solr.SpellCheckComponent" name="suggest">
  <lst name="spellchecker">
    <str name="name">suggestDictionary</str>
    <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
    <str name="lookupImpl">org.apache.solr.spelling.suggest.fst
      .FSTLookupFactory</str>
    <str name="field">suggest</str>
    <float name="threshold">0.</float>
    <str name="buildOnCommit">true</str>
  </lst>
</searchComponent>
```

Spell-checker 组件需要一个完整的查询 Term，然后根据它生成拼写建议。而 Autosuggest 组件只需要你提供一个字符前缀即可，正如我们前面所学的那样，Spell-Checker 组件是根据编辑距算法来计算两个 Term 之间的相似度，而 Autosuggest 不需要一个完整的 Term 也能工作，所以字符串编辑距在 Autosuggest 这里没有价值，Autosuggest 是采用通配符查询的方式来匹配相似 Term 的，遗憾的是，通配符查询性能是很差的，然而 Autosuggest 最大的要求就是查询响应速度快，显然我们需要另外一种方式来实现。Solr 内置的 Suggester 采用的是前缀树的数据结构来实现的，前缀树支持快速的前缀查找。比如当用户输入一个“at”，Solr 利用该数据结构能够快速查找出所有以“at”开头的 Term。当你的索引数据规模很大时，那么可能会返回大量的以“at”开头的 Term，所以我们还需要根据 Term 的出现频率进行从高到低的排序，我们不希望低频率或者罕见的 Term 作为拼写建议返回给用户。如果一个 Term 只在少数索引文档中出现，那么它就不是一个好的拼写建议。在我们上面的配置示例中，我们使用的是 org.apache.solr.spelling.suggest.fst.FSTLookupclass 类保证了前缀查询性能的高效快速，FSTLookupclass 内部使用的是基于 Finite State Automaton (简称 FST) 实现的数据结构，它支持固定的查找时间，与前缀字符串的长度无关。FSTLookupFactory 实现在构建 Spell-Checker 字典时有点慢，但内存占用相对较小，这对于构建大文本 Term 字典来说，FSTLookupFactory 是个不错的选择。当你添加了新的索引文档

时,你应该同时重建你的 suggest 字典,在配置中,我们通过 `buildOnCommit = true` 参数告诉 Solr 每次在提交新的索引文档之后都重新构建 suggest 字典。suggest 字典是缓存在内存中的,重新构建速度是很快的,所以重新构建 suggest 字典通常不是一个性能问题。

## 8.3 基于 N-Gram 实现 Autosuggest

Suggester 为指定的查询 Term 自动生成拼写建议提供了一种很好的解决方案,但是对于短语或者不分词的域比如我们示例中的 title 域, Suggester 就不好用了。在本章节中,我们将继续学习一种新的提供拼写建议的方式: N-Gram, 它支持对索引文档中的不分词域的域值提供拼写建议。在 solrpedia 示例中,我们可以对 title 域提供拼写建议并作为用户的最终输入,如果用户输入 river, 那么 Solr 的 Suggester 将会返回 rivers、riverdale 以及 riverside 这 3 个拼写建议,我们可以在页面上以下拉列表的形式将这 3 个拼写建议展示给用户,关键点是我们的 title 域是不分词的,即便我们的 title 域的域值为“Aariver”,当用户输入了“riv”,因为我们的 title 域是不分词的,使用 FST 去查找 title 域上以“riv”开头的 Term,“Aa river”并不符合要求,那么该如何实现类似这种需求呢?

正如我们在上面提及的那样,我们期望对不分词域或者短语也能智能的返回拼写建议,正因为不分词,所以可能域值中间包含了用户输入的前缀字符串,我们期望对这些中间包含用户输入前缀的域值也能返回作为拼写建议。即只要 title 域中的域值文本中包含了用户输入的“riv”这个前缀字符串,就应该返回该域值给用户作为拼写建议。实现这种需求的一种方式就是对 suggest 域使用通配符查询,比如“fiv\*”,但是采用这种方式会有两个问题:首先通配符查询的执行性能是非常之差的,特别是域值文本特别长的时候。再个就是通过通配符查询你没法评估找到的每个索引文档的相关性。比较好的实现方式就是对部分词域通过 copyField 复制到新的域上,然后对新的域在文本分词阶段进行 Edge NGram 处理。NGram 就是按照指定的长度生成连续邻近的字符串, N 表示生成的字符序列的长度,而 Edge NGram 表示每次都从边界开始 NGram 处理,所谓边界就是字符的两头左边还是右边,比如对“river”进行 2NGram,得到的就是“ri”、“iv”、“ve”、“er”,而对“river”进行 Edge 2NGram,得到的就是“ri”或者“er”。经过 NGram 处理后,我们的域值就能生成“riv”这样的 Term,从而与用户输入的“riv”匹配,从而作为拼写建议被返回给用户。

n-gram 处理有点类似于提前生成通配符查询可能的各种 Term,从而避免了通配符查询可能存在的性能问题,在查询时,使用“riv”前缀在 suggest 域上执行查询,会返回 suggest 域上所有进行 3-gram 处理后包含“riv”的域值。如果你使用 EdgeNGramFilter 对你的域进行分词处理,那么会生成所有可能存在的前缀并存入索引中,这也意味着使用 n-gram 会增大你的索引体积,因为你经过 n-gram 处理后原来的每个 Term 又生成了 M 个新的 Term 并被索引。因此在对域进行 n-gram 处理时,需要切记,尽量只对域值比较短的域进行 n-gram 处理,这样即便增大了索引体积,也不是很明显,还在可以控制的范围内。正

如在 schema.xml 中定义的那样，我们定义了一个新的域类型 text\_suggest\_ngram，用于支持对域值的 n-gram 处理，具体配置示例如下所示：

```
<fieldType name="text_suggest_ngram"
  class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.EdgeNGramFilterFactory"
      maxGramSize="10" minGramSize="2"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
  </analyzer>
</fieldType>
```

从上面配置你不难发现，我们仅仅只在索引阶段进行了 n-gram 处理，在查询阶段并没有配置 EdgeNGramFilterFactory。因为不需要对用户查询时输入的前缀字符串进行 n-gram 处理。

显然你并不希望返回的拼写建议是域值进行 n-gram 处理后生成的字符序列，你肯定希望的是返回的拼写建议是整个域的域值文本。如何实现，请看下面的配置示例：

```
<requestHandler name="/suggest_topic"
  class="org.apache.solr.handler.component.SearchHandler">
  <lst name="defaults">
    <str name="wt">json</str>
    <str name="defType">edismax</str>
    <str name="rows">10</str>
    <str name="fl">title</str>
    <str name="qf">suggest^10 suggest_ngram</str>
  </lst>
</requestHandler>
```

首先我们定义了一个新的 SearchHandler 便于与之前的区分开，这里对我们的 suggest 查询采用了 edismax 查询解析器，这使得可以同时多个域上执行查询，并为每个查询域设置不同的权重值，这里我们将 suggest 域的权重值设置为 10，表示优先返回 suggest 域上匹配的，然后最终返回的拼写建议是 title 域的域值文本。如上进行配置之后，你执行如下查询示例进行测试，即可实现根据前缀查询返回某个域的完整域值而不是分词后的某个 Term

作为拼写建议:

```
http://localhost:8080/solr/solrpedia//suggest_topic?q=riv
```

最终返回的结果如下所示:

```
"response":{"numFound":95,"start":0,"docs":[
  {
    "title":"Riverdale"},
  {
    "title":"Aa River"},
  ...// 省略
}]}
```

在结束本章之前, 我们还需要了解另一种基于用户行为实现拼写建议的方式。

## 8.4 基于用户行为实现 Autosuggest

之前拼写建议采用的方式虽然实现了需求, 但是并没有考虑用户输入一个查询关键字的次数。Solr 内置的 Suggester 适用于对指定的查询 Term 返回拼写建议, 但用户输入的查询关键字不可能都存在于索引中。基于 n-gram 实现的 Suggester 适用于对索引文档中的多个域基于查询 Term 前缀进行 suggest, 在这两种实现方式中, 都没有考虑用户的过去查询行为, 比如用户输入次数比较多的查询关键字应该优先返回作为拼写建议。比较好的实现方式应该是从一个最近查询次数比较多的关键字数据集中返回拼写建议。如果用户输入“王宝强”, 那么此时应该优先返回“王宝强离婚”、“王宝强马蓉”之类的查询关键字作为拼写建议, 而不是返回“王宝强大闹天竺”, 因为时下热点事件就是“王宝强离婚”, 用户根据那些关键字搜索的次数比较多。

我们之前的实现方式都是基于主索引库来进行 suggest 查询返回拼写建议的, 而基于用户行为的拼写建议实现方式需要对用户的查询行为记录、采集、解析用户日志构建另外一个索引库, 那么你需要开发一个工具来分析用户的查询行为日志并计算每个用户查询的流行度评分。每当用户发起了一个查询请求, 那么你的日志分析工作也要随之进行保证流行度评分能实时更新。当我们将根据用户行为构建的新索引库进行 suggest 查询返回拼写建议时, 需要按照每个关键字的流行度评分从高到低进行排序返回。

至于用户查询日志记录就留给读者自己去实现了, 为了简单起见, 假设我们拥有如表 8-2 展示的数据来表示用户查询日志数据。

表 8-2 用户查询日志数据示例

查询关键字	最后一次执行时间	频率 (最近一个月内)
Prince William	August 10, 2013	20000
Fort William	July 15, 2013	20005
William and Mary	July 22, 2013	19995

首先我们需要创建一个新的 Core 来存放我们的用户查询行为日志数据，新 Core 名称为 `solrpedia_instant`，请用户从随书源码中获取新 Core 相关文件并添加到 Solr 中。其中 `schema.xml` 定义如下所示：

```
<field name="id" .../>
<field name="query" type="string" indexed="false" stored="true"/>
<field name="query_ngram" type="text_suggest_ngram"
indexed="true" stored="false"/>
<field name="popularity" type="tfloat"
indexed="true" stored="true"/>
<field name="last_executed_on" type="tdate"
indexed="true" stored="true"/>
<field name="_version_" .../>
```

请注意，`query_ngram` 域我们使用的是 `text_suggest_ngram` 域类型，即会对字符串进行 N-Gram 处理。Core 添加成功后请将用户查询行为日志测试数据添加到新添加的 `"solrpedia_instant"` Core 中，测试数据如下所示：

```
[
  {
    "id" : "1",
    "query" : "Prince William",
    "last_executed_on" : "2013-08-10T00:00:00Z/DAY",
    "popularity" : 20000
  },
  {
    "id" : "2",
    "query" : "Fort William",
    "last_executed_on" : "2013-07-15T00:00:00Z/DAY",
    "popularity" : 20005
  },
  {
    "id" : "3",
    "query" : "William and Mary",
    "last_executed_on" : "2013-07-22T00:00:00Z/DAY",
    "popularity" : 19995
  }
]
```

请读者获取随书源码后直接运行 `IndexUserActivity` 类导入测试数据，运行完成后如果没有抛出异常，则表明测试数据导入成功。

那么思考一下，当用户输入多少个字符时，我们就应该启用用户的查询行为数据来返回拼写建议呢？之前的实现方式是，当用户输入的字符长度  $\geq 2$ ，就启用拼写建议，同时后续每多输入一个字符就即时更新拼写建议。但是基于用户行为的查询，如果用户输入的字符很短就开启用户的查询行为数据来返回拼写建议，可能返回的信息不是很精确，比如当用户输入“so”这两个字符时，我们无法得知用户是想要搜索“solr”，还是“sohu”，因



此只有当用户输入“solr”时，我们才可以让用户返回“solr wiki”、“solr tutorial”、“solr book”等热门的搜索关键字，这样返回的拼写建议才精准，才会让用户觉得你懂他。也就是说基于用户查询行为返回拼写建议不适合于输入字符长度太短的情况下启用，不是不能返回拼写建议，主要是考虑用户使用体验问题，因此当用户输入字符很短时先启用我们之前的实现方式来返回拼写建议，当用户输入的字符足够长了比如4个字符长度（具体长度自己控制调整），此时可以开启基于用户的查询行为数据来返回拼写建议，如果基于用户的查询行为数据来返回拼写建议得到的是空结果集，那么此时可以再切换回之前的基于主索引库的实现方式。

基于导入的测试数据可以得知，假如用户输入“willia”，我们应该将时下比较热门的搜索关键字“Fort William”作为拼写建议返回给用户，下面的查询示例演示了如何基于用户查询行为数据所在 core 进行查询：

```
http://localhost:8080/solr/solrpedia_instant/select?
q=query_ngram:willia&sort=popularity_desc&rows=1&fl=query&wt=json&indent=true
```

对我们的第二个索引库根据“willia”搜索关键字进行查询，返回了关于“willia”搜索关键字的3个历史比较热门的搜索关键字，并且是根据 popularity 热门指数从高到低排序的。假如我还想考虑按照这个搜索关键字的时效性来排序呢，比如某个搜索关键字的热门指数是挺高的，但是这个热门搜索关键字代表的热门事件都过去N年了，已经没有时效性了，我们应该尽量将当下比较热门的搜索关键字优先返回给用户，那我们该如何判断一个搜索关键字的时效性呢？

我们在考虑一个搜索关键字的热门指数的时候，搜索关键字的时效性同样重要，通过搜索关键字的时效性我们能对最近比较热门的搜索关键字进行加权，因为过去的热门事件人们已经失去兴趣了。比如我们的示例数据中，搜索关键字“Fort William”的热门指数要比“Prince William”高5个点，但“Fort William”搜索关键字查询发生的时间比较久远，那么理所当然如果我们考虑搜索关键字的时效性，在用户输入“willia”的情况下，“Prince William”搜索关键字应该是一个更好的拼写建议，因为它离当前时间更近。那么为什么不把搜索关键字的时效性也考虑到热门指数的计算当中呢？这是因为每过去一天，搜索关键字的时效性也在改变，因此你需要每天对你的用户查询行为数据所在索引库里的所有索引文档进行重建，这对于搜索请求比较频繁的应用来说显然不合适。我们认为比较好的方式是对每个索引文档添加该热门搜索关键字最后一次执行时间的 field（域），并且使用 Solr 中的 Function 查询为搜索关键字的时效性来动态计算它的权重值。想象下，假如我们只保存最近30天之内的用户查询行为数据在我们的第二个索引库中，当我们每天处理用户查询日志的时候，我们只需要处理最近30天之内的数据，也就是说我们就不用每天去重建用户查询行为数据所在的索引库啦。在建立索引库的时候，我们只需要每间隔30天更新一次搜索关键字的最后一次更新时间即可。

我们示例中的 last\_executed\_on 域表示的就是搜索关键字的最后一次执行时间，你需要

每间隔一个月更新一次这个域的数据。同时在这一个月之内,该搜索关键字可能还会被用户搜索,它的热门指数可能会稍微有所增长,因此同时你还需要更新 popularity 域的值。现在让我们来看看,该如何根据 popularity 和 last\_executed\_on 域使用 Solr 中的 Boost Function (加权函数)来对一个索引文档进行加权,请看下面这个查询示例:

```
http://localhost:8080/solr/solrpedia_instant/select?
q={!boost b=$recency v=$qq}&
sort=score desc&
rows=1&wt=json&indent=true&
qq=query_ngram:willia&
recency=product(recip(ms(NOW/HOUR,last_executed_on),
1.27E-10,0.08,0.05),popularity)
```

上面的查询示例将“Prince William”作为 Top one 拼写建议返回给用户,尽管它的热门指数不是最高的,但是它离当前时间更近。boost function (加权函数)会依据时间的增长对索引文档的权重值进行慢慢衰减,下面对上面查询示例中的部分查询语法稍作解释:

```
qq=query_ngram:willia
```

这里表示定义了普通的域查询,对用户输入的搜索关键字“willia”在 query\_ngram 域上执行查询,然后将这个查询通过一个自定义变量 qq 缓存起来:

```
sort=score desc
```

显式的指定按照索引文档的最后相关性评分降序排列:

```
recency=product(
  recip(ms(NOW/HOUR,
    last_executed_on),
    1.27E-10,0.08,0.05),
  popularity)
```

首先 product 函数用于计算两个数值的乘积,搜索关键字的时效性权重是根据 Solr 中的 recip 函数计算的,它的函数定义如下:

```
recip(x,m,a,b) = a / (m*x + b)
```

x 参数表示索引文档的 age 年龄,参数 m, a, b 用于计算 x 的惩罚值。索引文档的年龄取的是 last\_executed\_on 域的域值表示的时间与当前时间之间的毫秒数之差。

```
q={!boost b=$recency v=$qq}
```

使用 Solr 中的 Boost Query Parser 来对匹配查询(使用参数 v 表示的)的索引文档设置权重,权重值根据 b 参数值 \$recency 变量表示的加权函数来计算得到,如图 8-4 所示。

Solr: 根据索引文档的年龄来加权

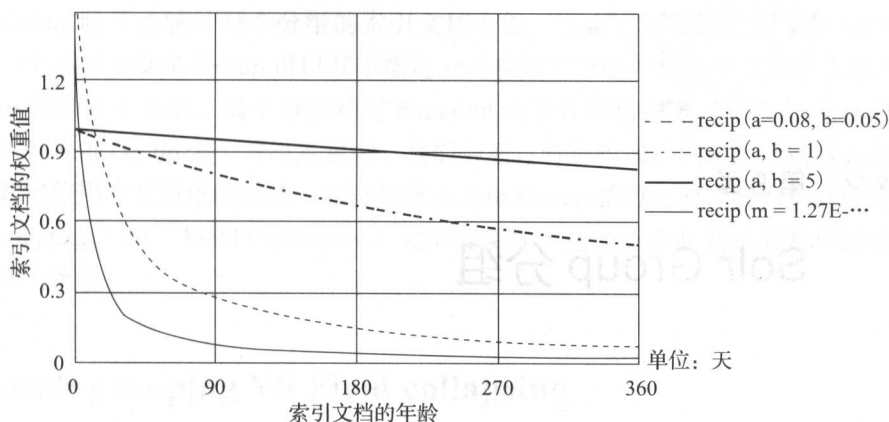


图 8-4 recip 加权函数在不同 a, b, m 参数下的权重值曲线图

## 8.5 本章总结

在本章中，我们学会了如何使用 Spell-Check 查询组件来处理用户输入搜索关键字时出现拼写错误的情况，我们还学会了如何实现为查询返回拼写建议来提升搜索程序的可用性以及用户体验。此外，我们还使用 Solr 内置的 Suggester 组件为用户返回拼写建议，Spell-Check 与 Autosuggest 是互补性的查询组件，两者结合在一起构成了强大的拼写建议解决方案。然后我们尝试使用了另外一种方式来实现对任意域或查询提供拼写建议，这里我们采用的是在创建索引的同时使用 N-Gram 处理生成更多的 Term，这样就能非常高效的使用前缀字符来返回拼写建议。最后讨论了如何实现基于用户查询行为来返回拼写建议。下一章中，我们将继续学习 Solr 中的 Group 分组功能。

## Solr Group 分组

通过第 9 章，你将可以学习到如下内容：

- ❑ 如何“删除”查询结果集中的重复索引文档；
- ❑ 如何在单一请求中对查询结果集返回多个分组；
- ❑ 如何实现分组查询的分页和排序；
- ❑ 如何根据指定域、任意查询、Function 函数动态计算值对索引文档进行分组；
- ❑ 如何实现基于 Group 分组返回的结果集进行 Facet 查询统计；
- ❑ 如何高效的实现 Group 分组查询。

当你使用搜索引擎时可能已经见过了类似“Field Collapsing”（即域折叠或者域收缩）功能，如果搜索引擎告诉你有很多结果匹配了，但是只是显示了部分结果，那么有可能你已经感受到了“Field Collapsing”功能。通常，“Field Collapsing”功能还会提供一个链接给用户，用户单击后会显示展开后的完整查询结果集。什么？还是不清楚什么是“Field Collapsing”？举个例子吧，淘宝我想大家都上过吧，当你在淘宝上搜“衣服”，衣服会有很多品牌和款式，对于同一个品牌同一款式的衣服可能还会很多不同的颜色（红、蓝、绿、黄、青、黑、白、灰等）和尺码（S、M、L、XL 等），你肯定不希望同一个品牌同一款式的衣服因为不同颜色不同尺码就占满了整个屏幕，用户大都希望对于同一个品牌同一款式的衣服就显示一条即可。但是对于 Solr 而言，不同颜色不同款式在索引中其实是不同的 Document，即是按照多个索引文档来存储的，“Field Collapsing”功能就是用于解决此类问题，它会按照指定值对索引文档进行收缩折叠，对于同一个值会落入同一个分组最终只返回 top one，这就是“Field Collapsing”。

此外，Solr 中的结果集分组功能还提供了其他有用的特性。一般在 Solr 中对查询结果

集进行分组显示你可以采用 Facet 方式，但是 Facet 仅仅只返回每个维度下的统计数字，而 Solr 中的 Group 除了会统计每个分组的索引文档个数，还会返回每个分组下匹配的索引文档。另外一个不同点就是 Group 可以基于指定 sort 参数对分组中的索引文档进行排序。Solr 中的 Group 可以基于域值、基于域值经过 Function 动态计算后得到的值、任意查询进行分组查询。尽管 Solr Group 乍一看有点复杂，我们会通过各种查询示例来演示这些大部分非常有用，但其实使用非常简单的功能。在开始进入 Solr Group 学习之前，我们有必要先弄清楚“Result Grouping”和“Field Collapsing”之间的区别，这也是 Solr 新手经常容易感觉迷惑和混淆的两个概念。

## 9.1 Result grouping VS Field collapsing

Solr 新手经常会问到的一个问题就是：为什么 Solr 中的分组会有两个名称：Result grouping 和 Field collapsing，两者有什么区别？

Field Collapsing 用于解决对于索引文档中大部分域的域值相同，只有个别域的域值不同的情况下只返回一个索引文档，正如我们上面所举的衣服颜色尺码例子那样。早期版本中，Field Collapsing 这个功能是为扩展 Solr 而开发，有个非正式的名称叫做“Field Collapsing Patch”。后来这个功能越来越通用了就被纳入到正式版本中。而 Result Grouping 通常表示更常用的结果集分组功能。尽管 Field collapsing 在指定域的多个重复值上返回单个索引文档时需要 Result grouping 的支持，但 Field collapsing 其实也支持对于单个查询返回多个结果集或者多个分组。

Result grouping 相比 Field collapsing 而言，它表示一种更通用的使用场景即结果集分组，而 Field collapsing 更多是对传统的结果集分组功能的一个扩展。Field collapsing 还有一种更高效的实现方式（使用 Collapsing Query Parser 实现），但这种方式使用起来有一些限制，它要求在进行 collapsing 之前必须先对索引文档进行排序，但是在某些场景下，这种方式会很有用，稍后会详细讲解。本章主要目标是 Solr 的结果集分组功能即 Result Grouping。

## 9.2 按照指定域分组

假设你正在访问一个电子商务网站，你输入一个搜索关键字，它可能会返回很多结果集，但可能都是同一类的商品仅仅只是颜色尺码不同或者仅仅只是发货方不同，这对于你来说或许并不是你所期望的，你会希望对于同一类商品显示一个就够了。正由于有了 Field collapsing 的存在，使得这种需求的实现变得简单，同时它也允许你返回的结果集中的每个索引文档具有多样化或者差异性。假如你输入搜索关键字“spider man”（蜘蛛侠），如果返回的结果集顶部是“The Amazing Spiderman—2012”，但是随后有 100 个重复项，且假设

这 100 项结果的相关性评分都是相同，那么前 100 项结果应该只返回顶部那个就行了。那么你可以对返回的结果集按照名称分组，那么“Amazing Spider-man—2012”就只会显示一次。如果你将同一个商品在一个页面重复展现 10 次、100 次，这显然会降低用户继续搜索“spider man”的欲望。在大部分情况下，使用 Result grouping 来收缩折叠查询结果集可以改善用户体验。

为了方便演示 Solr Group 相关功能，你需要先从随书源码中获取示例代码和相关配置文件，然后创建“ecommerce”Core 并导入测试数据，测试数据文件在 example-docs\ch09\documents 目录下。你只需要运行 IndexECommerce 类即可完成测试数据的导入。确保一切准备就绪之后，让我们开始 Solr Group 的学习之旅。首先请执行如下所示的查询示例：

```
http://localhost:8080/solr/ecommerce/select?
fl=id,product,format&
sort=popularity asc&
q=spider-man
```

返回的查询结果部分显示如下所示：

```
"response":{"numFound":18,"start":0,"docs":[
  {
    "id":"4",
    "format":"dvd",
    "product":"The Amazing Spider Man - 2012"},
  {
    "id":"5",
    "format":"blu-ray",
    "product":"The Amazing Spider Man - 2012"},
```

从返回的查询结果可以看出，部分结果中的 product 域的值是相同的，只是 format 值不同，这跟我们之前举的例子：“衣服的品牌款式等信息都相同只是颜色尺码不同”差不多，同一个商品因为某些值不同而被显示多次这是很糟糕的用户体验。接下来我们尝试开启 Group，请执行如下查询示例：

```
http://localhost:8080/solr/ecommerce/select?
fl=id,product,format&
sort=popularity asc&
q=spider-man&
group=true&
group.field=product&
group.limit=1
```

返回结果部分显示如下所示：

```
"product":{"
  "matches":18,
  "groups":[{
    "groupValue":"The Amazing Spider Man - 2012",
    "doclist":{"numFound":2,"start":0,"docs":[
```



```
{
  "id": "4",
  "format": "dvd",
  "product": "The Amazing Spider Man - 2012"}}
},
...// 其他省略
```

关于 Solr 中的 Group 有几个关键点值得注意一下：首先，需要明白 Solr 中的 Group 功能必须通过指定 `group = true` 参数来显式开启。然后你需要指定 `group.field` 参数，表示对哪个域进行分组。根据指定域分组后，该域的域值相同的索引文档会落入同一分组内，而 `group.limit` 参数用于指定同一分组内最多返回多少个索引文档，这里我们设置为 1 即表示对于每个分组内只返回 1 个索引文档。当你想移除所有“重复”的索引文档时，`group.limit` 参数设置为 1 对你来说可能会有意义。注意，这里说的“重复”只是我们主观感觉上的重复，其实索引文档的个别域的域值还是不同的，严格意义上来说两者并不是完全重复的，只是看起来像是“重复”了。

返回的 `groupValue` 属性表示每个分组，值为分组域的每个唯一域值，“`doclist`”下的 `numFound` 属性表示当前分组下有多少个索引文档。

如果你仅仅只想要移除“重复”的索引文档，并不需要其他额外的分组信息，比如每个分组下匹配的索引文档总数，那么你可以通过设置 `group.main = true` 参数来合并每个分组的结果形成一个扁平化的列表并最后在主结果集展示区“`docs`”部分显示，下面的查询示例演示了 `group.main` 参数的使用：

```
http://localhost:8080/solr/ecommerce/select?
fl=id,product,format&
sort=popularity asc&
q=spider-man&
group=true&
group.field=product&
group.main=true
```

返回的查询结果集部分如下所示：

```
"response":{"numFound":18,"start":0,"docs":[
  {
    "id": "4",
    "format": "dvd",
    "product": "The Amazing Spider Man - 2012"},
  {
    "id": "6",
    "format": "dvd",
    "product": "Spider Man - 2002"},
```

从上面返回的查询结果集来看，就像我们没有使用 `group` 查询似的。但是开启 `group.main` 参数有个最大缺点就是没有返回每个分组下匹配的索引文档的总数，但是如果每个分

组下匹配的索引文档的总数这个数值对你的搜索程序来说并不关心的话，那么你就可以将 `group.main` 设置为 `true`，同时不用解析处理两种格式的查询结果集。而且开启 `group.main` 参数之后，你还无法获取每个分组的名称即 `groupValue` 值，但是这个值通常可以通过分组的域值推断出来。开启 `group.main` 参数另外一个缺点就是它只支持单个分组，使用 `Result Grouping` 来实现 `Field collapsing` 最大的缺点就是它的执行性能比标准的查询请求还差。值得庆幸的是，自 Solr 4.6 开始，提供了一种全新的基于 `Collapse Query Parser` 实现的更高效的 `Field collapsing`，它会在主结果集展示区“docs”部分显示分组查询结果集，就像你使用了 `group.main = true` 参数。

目前为止，我们还仅仅只是对单个域进行分组，其实 Solr Group 还支持对多个域进行分组，比如 `group.field = type&group.field = format`，如果同时还指定了 `group.main = true`，那么 Solr 会只返回最后一个分组。你还可以指定 `grouping.format` 参数来设置分组结果集的输出格式，`group.format = simple` 会类似 `group.main` 那样以扁平化的列表形式展示分组内的每个索引文档，但此时不是在主结果集展示区“docs”部分显示，具体请大家执行如下查询示例去感受下：

// group.format 参数的使用

```
http://localhost:8080/solr/ecommerce/select?
```

```
fl=id,product,format&
```

```
sort=popularity asc&
```

```
q=spider-man&
```

```
group=true&
```

```
group.field=product&
```

```
group.format=simple
```

// group.main 参数的使用以及多个域分组

```
http://localhost:8080/solr/ecommerce/select?
```

```
fl=id,product,format&
```

```
sort=popularity asc&
```

```
q=spider-man&
```

```
group=true&
```

```
group.field=type&group.field=format&
```

```
group.main=true
```

// 默认多个域分组的结果集输出格式测试

```
http://localhost:8080/solr/ecommerce/select?
```

```
fl=id,product,format&
```

```
sort=popularity asc&
```

```
q=spider-man&
```

```
group=true&
```

```
group.field=type&group.field=format
```

到此，你已经了解了如何通过 `group` 分组查询来收缩折叠结果集，从而达到删除重复索引文档的目的（其实并不是真正的删除，只是不显示给用户而已）。你也了解了分组查询结果集的 3 种输出格式：默认格式、`group.format = simple` 表示的简单分组输出格式以及

`group.amin = true` 表示的显示在主查询结果集展示区的单一收缩组，请大家通过上面的几个查询示例自己去观察意会它们之间区别。

本章的剩余部分，将会继续介绍 Solr 分组中更高级的用法，比如如何在一个单独查询中为每个分组返回多个文档。

## 9.3 每个分组返回多个文档

Solr 的 Group 分组查询实际上并不仅仅只是用来对每个域的唯一值只返回单个索引文档（前提是 `group.limit = 1`），回顾我们之前章节中的 e-commerce 演示示例，假设仅仅只是收缩折叠重复的索引文档，我们可以保证每个分组都返回不超过固定数目的索引文档。当分组数目很多时，我们可以通过 `rows` 参数来限制返回的分组总个数，`start` 表示分组返回的起始索引位置（从零开始计算），这里跟普通的 Query 查询分页有点类似，这里是对返回的所有分组进行分页。具体请看下这个查询示例：

```
http://localhost:8080/solr/ecommerce/select?
q=spider-man&
fl=id,product,format&
sort=popularity asc&
group=true&
group.field=type&
group.limit=3&
rows=5&
start=0&
group.offset=0
```

从上面的查询示例可以得知以下几点：首先你需要注意 `group.limit` 参数表示限制每个分组返回的索引文档最大个数，但并不表示每个分组必须返回 3 个索引文档，有可能该分组下只有 2 个索引文档。`group.limit` 参数只是设置一个上限值。第二点，你需要注意的是 `rows = 5` 参数控制的不是有多少索引文档返回，而是最多有几个分组被返回，在默认的分组输出格式中，`rows` 和 `start` 参数是应用于每个分组，而不是每个分组内的索引文档。这两个参数搭配在一起使用其实就是对返回的所有分组进行分页。`group.offset` 参数用于控制每个分组内的索引文档的起始偏移量，比如某个分组下有 10 个索引文档，但 `group.limit` 设置为 3，那么默认该分组下只会返回 0, 1, 2 这 3 个索引文档，如果 `group.offset` 设置为 2，即表示每个分组从第 2 个位置开始返回索引文档。注意，这里的 `offset`（偏移量）是从零开始计算的，所以此时返回的 3 个索引文档就是第 2, 3, 4 这 3 个索引文档，而且还要注意，`group.offset` 参数值不能大于等于每个分组下所有索引文档的总个数。

最后比较重要的一点就是分组中的排序问题。先假设分组查询未启用，那么默认所有的索引文档会按照相关性评分从高至低排序，当开启分组后，相当于在每个分组内部求索引文档评分最大值，直到所有分组内的评分最大值计算完之后，最后按照每个分组内的最大评

分从高至低进行排序。这就好比先在每个班里挑选成绩最好的学生，然后把每个班里成绩最好的学生派去参加竞赛然后按照最后成绩排名，每个参加竞赛学生最后的成绩排名代表他所在班级的排名。

在每个分组内部，索引文档也会进行排序，默认是按照 id 域从小到大进行排序。当你将 `group.limit` 参数设置为 1 且 `group.main = true` 或者 `group.format = simple` 时，那么默认就是按照 id 域从小到大进行排序。

## 9.4 按照 Function 动态计算值分组

除了能够按照指定域的唯一性域值进行分组之外，Solr 还支持两种分组方式。第一种有点类似于按照指定域进行分组，但是它允许你对指定域应用 Function Query 动态计算值，最后按照动态计算值进行分组。第二种就是按照 Query 进行分组，它允许同时执行多个 Query 并返回独立的结果集。

按照 Function 进行分组你需要指定 `group.func` 参数来完成，这里暂时不全面介绍 Solr 中所有的 Function，会留到后面专门讲解。下面这个查询示例演示了如何按照 Function 进行分组：

```
http://localhost:8080/solr/ecommerce/select?
fl=id,product,format&
sort=popularity asc&
q=spider-man&
group=true&
group.limit=3&
rows=5&
group.func=map(map(map(popularity,1,5,1),6,10,2),11,100,3)
```

在上面这个示例中，`group.func` 参数指定的函数尝试按照 `popularity`（流行度）将索引文档分 3 个等级，你会发现返回的分组结果集跟按照指定域进行分组是类似的，唯一的区别就是按照 Function 分组作用对象是 Function（函数）动态计算得到的值，而按照指定域进行分组的作用对象是分组域的每个唯一性的域值。这就有点类似于 SQL 语句里的按照 `count()/sum()/avg()` 这些函数动态计算值进行分组和按照指定字段分组的区别。这里的 `map(popularity, 1, 5, 1)` 其实就是对 `popularity` 域的域值进行一个映射，表示 `popularity` 域值在 `[1, 5]` 这个区间内的落入第 1 组，同理 `map(map(popularity, 1, 5, 1), 6, 10, 2)` 表示 `popularity` 域值在 `[6, 10]` 这个区间内的落入第 2 组，依此类推。

Function 是可以嵌套的，正如你在上面示例中所看到的那样，我们对 `map` 函数嵌套了 3 次，这意味着你可以结合多个函数灵活控制函数的动态计算值。如果按照函数分组对于你来说，还有太多限制的话，你还可以按照 Query 进行分组，这样你就可以将任意你指定的值进行分组。

## 9.5 按照任意 Query 分组

通过前面的章节了解到，我们可以通过 `group.field` 参数对指定的域进行分组，从而达到对分组域的每个特定值下匹配的索引文档进行收缩折叠。如果 `group.limit` 参数设置为 1，还可以实现对每个分组下重复索引文档进行“删除”。你除了可以对提前指定域的域值进行分组之外，还可以动态的对任意查询进行分组，你可以定义多个 Query，同时对多个 Query 进行分组，就好比对多个域进行分组。为了演示 Solr Group 的这种功能，让我们尝试对 3 个 Query 进行分组：一个查询匹配所有 `type = Movies` 的产品，一个查询所有包含“games”关键字的，一个查询所有包含“The Hunger Games”这个短语的。请动手执行如下查询示例：

```
http://localhost:8080/solr/ecommerce/select?
sort=popularity asc&
fl=id,type,format,product&
group.limit=2&
q=*&
group=true&
group.query=type:Movies&
group.query=games&
group.query="The Hunger Games"
```

从上面的查询示例我们可以获取到以下 3 点心得：

- ❑ 对于任意的分组查询，不管是 `group.field` 或 `group.func` 还是 `group.query`，你都可以返回多个分组；
- ❑ 你可以在原查询基础之上执行多个分组子查询，这里的多个分组都是一个分类上分组，并不是多个分组的叠加，这跟 SQL 里的 `group by a,b` 不同。多个分组之间是独立的，之间并没有联系；
- ❑ 按照某个域分组，那么某个索引文档必定只可能属于一个分组，但是如果按照多个 Query 分组，那么某个索引文档可能属于多个分组。

## 9.6 Group 的分页与排序

在 Solr 的普通查询中，我们使用 `rows` 参数来告诉 Solr 最多返回多少个索引文档，然而在 Group 分组查询中，这里有一点复杂，`rows` 参数到底限制的是什么呢，你是想限制每个分组下返回索引文档个数还是所有分组的个数，还是所有分组总共返回的索引文档总个数？相似的问题还有当你在分组查询中使用 `start` 参数进行分页以及使用 `sort` 参数对分组内的索引文档进行排序，这些参数表示什么含义呢？Group 查询的相关参数，如图 9-1 所示。

在 Group 分组查询中，`rows` 参数用于限制返回的分组总个数，`start` 参数用于控制对分组进行分页时第一个分组的起始位置（从零开始计算）。因此在 Group 分组查询中，将 `start` 参数与 `rows` 参数搭配使用，可以实现对分组的分页查询。`sort` 参数用于控制如何对每个分

组进行排序，默认是基于当前分组下评分最高的索引文档的分数从高到低排序，而不是控制分组内每个索引文档的排序。

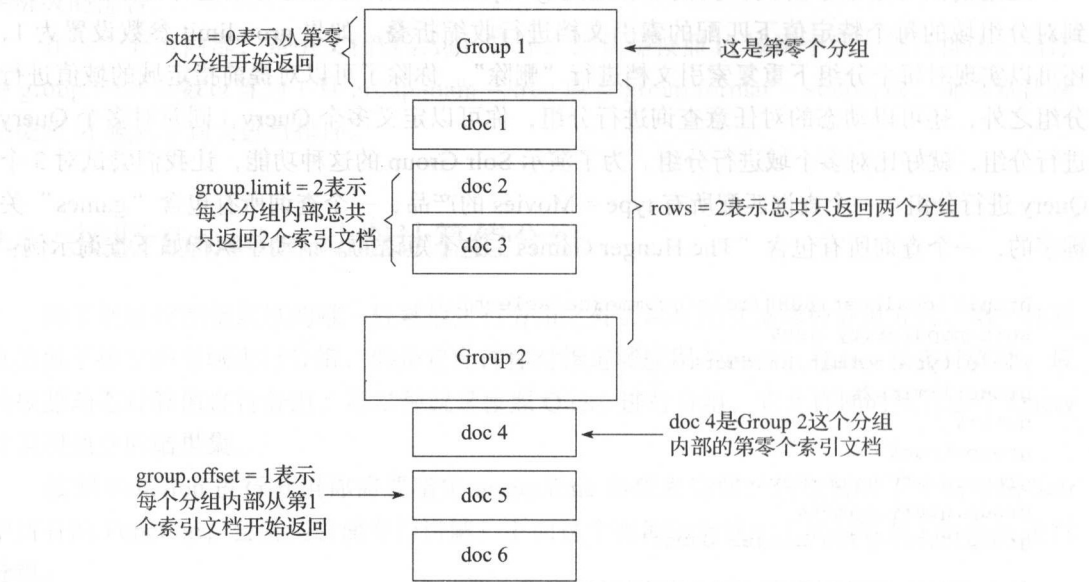


图 9-1 Group 查询中的相关参数

group.limit 参数用于指定每个分组内最多返回多少个索引文档，group.offset 参数用于分组内返回的索引文档的起始位置（从零开始计算），将此参数与 group.limit 参数结合使用，你可以实现对分组内的索引文档进行分页。而 group.sort 参数允许你对分组内部的每个索引文档进行排序。

也许理解分组查询中的分页、排序最简单的方式就是将分组看作 Solr 中普通的索引文档，你可以在标准的 Solr 查询中对返回的索引文档进行分页、排序，也可以对分组查询中的所有分组以及每个分组内部的索引文档进行分页、排序，为此 Solr Group 提供了 group.limit、group.offset 以及 group.sort 等参数帮助你控制如何显示分组以及每个分组内部的索引文档。

## 9.7 Group & Facet

默认 Facet 查询统计是基于 q 参数的查询结果集的，而不是 Group 分组查询返回的结果集，这意味着不管你是否开启分组查询，Facet 查询统计返回的结果都是一样的。请看下面的查询示例：

```
http://localhost:8080/solr/ecommerce/select?
fl=product,type,score&
```



```
group=true&
sort=popularity asc&
q=*&
facet=true&
facet.mincount=1&
fq=type:Movies&
facet.field=type&
group.field=product
```

返回的查询结果部分如下所示：

```
"product":{
  "matches":11,
  "groups":[{
    "groupValue":"The Hunger Games",
    "doclist":{"numFound":1,"start":0,"maxScore":1.0,"docs":[
      {
        "type":"Movies",
        "product":"The Hunger Games",
        "score":1.0}}
    ]},
  {
    ...// 省略
  }
}
"facet_counts":{"facet_queries":{},
  "facet_fields":{"type":[
    "Movies",11]}}
```

从上面返回的结果集你应该注意到尽管 Filter Query type: Movies 匹配了 11 个索引文档，但按照 product 域分组查询后只返回了 6 个“索引文档”，这是因为这 11 个索引文档中有些索引文档的 product 域的域值相同仅仅只是 type 域的域值不同而已，这里的 6 个是按照 product 域分组后有 6 个分组。另一个你需要注意的是 Facet 查询统计仍然是基于 q 和 fq 参数的，而不是基于 group 分组查询。此外你还需要注意的是每个分组内的 numFound 属性值虽然统计是正确的，但是每个分组下返回的索引文档数与 numFound 值并不一致，这是因为 group.limit 默认为 1，设置为 -1 就全部返回了。最后需要注意的是，Group 分组查询和 Facet 查询统计都是基于 q 和 fq 参数的，两者没有任何联系。此时可以设置 group.facet = true，请看下面的查询示例：

```
http://localhost:8080/solr/ecommerce/select?
fl=product&
group=true&
sort=popularity asc&
q=*&
facet=true&
facet.mincount=1&
group.format=simple&
```

```
fq=type:Movies&
facet.field=type&
group.field=product&
group.facet=true
```

返回的查询结果如下所示：

```
"product":{
  "matches":11,
  "doclist":{"numFound":11,"start":0,"docs":[
    {
      "product":"The Hunger Games"},
    {
      "product":"The Amazing Spider Man - 2012"},
    {
      "product":"Spider Man - 2002"},
    {
      "product":"Spider Man 2 - 2004"},
    {
      "product":"Top Gun"},
    {
      "product":"A Beautiful Mind"}}
  ]}},
"facet_counts":{
  "facet_queries":{},
  "facet_fields":{
    "type":[
      "Movies",6]},
```

从上面返回的结果集我们可以发现此时 Facet 查询统计得到的数字是 6，而我们的 Group 查询返回的分组总个数也是 6，说明此时我们的 Facet 查询统计实际是分组总个数了。换句话说，此时 Facet 是基于 Group 查询后返回的结果集进行统计。类比到关系型数据里 SQL Group By 语法，`group.facet = true` 其实就相当于 Group By 后返回的结果集条数，举个例子，假如我们一张 user 表，有 id、name、province 3 个字段，我们执行如下 SQL 语句：

```
select count(id),province from user group by province
```

上面的 SQL 语句其实就是对用户按照省份进行分组，返回的结果如图 9-2 所示。

正如你所看到的那样，该 SQL 一共返回了 6 个分组，第一列为每个分组下的记录总条数，而我们的 `group.facet = true` 参数开启之后，统计的其实就相当于分组的总个数，即我们这里 Group By SQL 语句返回的记录总条数。

信息	结果1	概况	状态
count(id)	province		
48	Beijing		
1	Hebei		
1	Hubei		
2	Liaoning		
1	Shanxi		
1	Zhejiang		

图 9-2 SQL 中对用户按省份分组统计

默认通过 `group.facet = true` 设置是全球生效的，如果不想全局生效，你只想单独对某个域进行设置，那么可以这样指定，语法如下所示：

```
f.<fieldName>.group.facet=true
```

这也是 Solr 中很多请求参数所支持的为单独某个域局部设置的语法。这里的 `<fieldName>` 表示你想应用 `group.facet = true` 的 Facet 域名称即 `facet.field` 的某个域名称。

遗憾的是，当你按照多个域进行分组时，Facet 查询统计并不会统计每个分组域返回的分组总个数，只会统计第一个分组域 Group 之后返回的分组总个数。具体请执行如下查询示例自己感受下：

```
http://localhost:8080/solr/ecommerce/select?
fl=product&
group=true&
sort=popularity asc&
q=*:*&
facet=true&
facet.mincount=1&
group.format=simple&
fq=type:Movies&
facet.field=type&
group.field=product&
group.field=format&
group.facet=true
```

## 9.8 Group 分布式查询

在使用 Solr 的 Group 分组查询时，需要考虑下的一个问题就是我们该如何进行 Group 分布式查询。不像标准的 Solr 查询，Group 分组查询不能完全在分布式模式下工作，更准确地说，它是运行在伪分布式模式下。结果集聚合是在分布式模式下运行的，但是聚合需要的每个结果集却是分别在每个本地 Core 上单独进行 Group 计算而来的。

这样为什么会有问题？因为你分组查询依赖的值可能是随机分布在多个 Solr Core 上，这样你 Group 分组统计的数字可能会不准确，比如你对产品按照制造商进行分组查询，那么得到的总分组个数可能会约等于实际每个 Core 上统计的分组总个数。当且仅当，你的索引文档按照制造商这个域进行分区时，你会得到正确的分组个数。因为每个分组只会存在于一个唯一的 shard 上。当在分布式模式下使用 Solr Group 功能时，你需要时刻记住这点，你需要设置 `group.ngroups = true`，以保证能够返回分组的总个数。如果你的索引数据没有按照你分组的域进行分区，并且你执行的是分布式搜索，那么返回的分组总个数可能仅仅为粗略值。

除了需要考虑数据分区的问题之外，某些 group 参数在分布式模式下也不支持，比如 `group.truncate`、`group.func`。`group.truncate` 参数如果设置为 `true`，则表示 Facet 查询统计会基于当前查询匹配的每个分组中相关性最高的文档进行统计数量。`group.func` 参数表示基于 Function Query 动态计算值进行 Facet 查询统计，参数值一般是表示 Function Query 的查询表达式。

最后你需要注意的是，Solr Group 分组查询不支持多值域，即你不能在一个多值域（即

multiValue = true) 上进行分组查询。虽然 Solr Group 分组查询支持分词域 (即域类型为 TextField), 但 Solr 并不会将分词后得到的每个 Token 作为 GroupValue, 你也不能选择按照哪个 Token 来分组, 而且 Solr 也不保证会按照分词后得到的每个 Token 进行分组, 有可能会丢弃部分 Token。因此对分词域进行分组查询没有太大意义, 尽管对分词域进行分组查询不会报错, 但最后得到的结果并不可靠, 因此, 一般建议你需将需要分组的域设置为不分词的单值域。

## 9.9 Group 缓存

尽管 Solr Group 查询功能很强大, 跟 Solr 标准查询相比, 查询速度那是相当的慢。对大数据量的索引文档按照指定域进行分组查询比不分组查询要花费更多的时间。

为了提升 Solr Group 分组查询的性能, 你可以通过指定 group.cache.percent 参数来为分组查询启用缓存。group.cache.percent 参数的默认值为零, 取值范围为 [1, 100] 则表示为 Group 启用缓存。Solr Group 内部会分两次查询来执行, group.cache.percent 参数表示第一次查询匹配的索引文档应该被缓存的百分比, 同时, Group 缓存还会附带缓存索引文档的相关性评分, 缓存第一次查询是为了提升第二次查询的性能。group.cache.percent 参数值设置的越大, 那么你的 Group 查询占用的内存将会更多, 所以你应该设置不同的百分比来测试出以尽量少的内存占用来获取最快的执行速度。当你的第一次查询是 Boolean 查询、Wildcard (通配符) 查询或者 Fuzzy (模糊) 查询, 此参数能显著提升查询性能。

如果你只需要返回每个分组, 并不需要返回每个分组下包含的所有索引文档, 而且你也不需要分组进行排序, 那么你可以采用更高效的方式来实现 “field collapsing” (只返回每个分组即域折叠), 下一章节会做详细介绍。

## 9.10 使用 Collapsing Query Parser 实现高效的 Field Collapsing

Solr 中提供了一种 Collapsing Query Parser (对应 CollapsingQParserPlugin 类), 它允许你将查询结果集为每个唯一值折叠收缩成一个单一的结果, 而且使用起来还不复杂。使用语法如下所示:

```
/select?q=*:*&fq={!collapse field=fieldToCollapseOn}
```

这个查询会为 field 属性指定的域下每个唯一值只返回一个索引文档, 而且返回的那个索引文档是包含该域值的所有索引文档中评分最高的。你还可以通过设置 min/max 属性来控制返回的索引文档, 比如:

```
/select?q=*:*&fq={!collapse field=fieldToCollapseOn min=numericFieldName}
/select?q=*:*&fq={!collapse field=fieldToCollapseOn max=numericFieldName}
/select?q=*:*&fq={!collapse field=fieldToCollapseOn max=sum(field1, field2)}
```

此外, Collapsing Query Parser 通过 nullPolicy 策略以多种方式支持对域值为空的索引文档的处理, nullPolicy 策略用于决定如何处理这些包含了空值的索引文档, 默认包含 3 种策略: ignore、expand、collapse。ignore 表示移除所有包含空值的索引文档, ignore 也是 nullPolicy 的默认策略。使用语法如下所示:

```
/select?q=*&fq={!collapse field=fieldToCollapseOn nullPolicy=ignore}
```

当 nullPolicy = collapse 时, 那么所有包含空值的索引文档会被分为一组, 并最终折叠收缩为一个文档。如果你想要返回所有包含空值的索引文档, 那么你可以指定 nullPolicy = expand。

理论上讲, Collapsing Query Parser 返回的结果集与你使用 Group 分组查询 (group = true) 且指定了 group.main = true、group.limit = 1、sort = score desc (或者指定按照指定的数字域排序) 返回的结果集是相似的。

但使用 Collapsing Query Parser 时有一个限制你需要注意: Group 分组查询首先会在文档折叠收缩之前对所有匹配的索引文档进行排序, 但是 Collapsing Query Parser 值考虑单一因素 (相关性评分或者 min/max 值), 因此对 Collapsing Query Parser 设置 sort 参数无效。此外你还可以对 min、max 属性应用 Function 函数, 通过 Function 函数动态计算值来确定 min 或 max 值, 从而确定 top one 索引文档, 使用示例如下所示:

```
/select?q=*&fq={!collapse field=fieldToCollapseOn max=sum(product(field1, 1000000), cscore())}
```

在上面这个示例中, 我们先按照 field1 域降序排, 然后按照 cscore 函数计算值降序排, cscore 函数用于在索引文档折叠收缩之前计算索引文档的分数。

## 9.11 Solr Group VS SQL Group by

在本章中, 我们已经学习了如何使用 Solr 中的 Result Grouping (结果集分组) 和 Field Collapsing (域折叠收缩) 功能, Result Grouping 功能有点类似于 SQL 里的 Group By, 但是 Solr 中的 Result Grouping (结果集分组) 不仅仅支持对某个域进行分组, 还是对任意的 Query、任意的 Function 在查询时动态计算值进行分组。通过将 group.limit 参数设置为 1 可以实现 Field Collapsing (域折叠收缩) 功能, 看起来像是将域值重复的索引文档“删除”了。你还可以通过指定多个 group.field 参数支持在单个查询请求中执行多个域的分组查询, 但是注意, 这里并没有实现分组嵌套效果。何为分组嵌套? 还记得我们 SQL 里的 Group By 吗? 在 SQL 中假如我们按照两个字段进行分组, 我们可以这样写:

```
select * from table_name group by location, type
```

在 SQL 中是会首先按照 location 字段进行分组, 然后在每个分组内部再按照 type 字典

进行分组，返回的结果可能是类似下面这样的结构：

```
-Location X
  ---Type 1
    ----records
  ---Type 2
    ----records
-Location Y
  ---Type 1
    ----records
  ---Type 2
    ----records
```

而 Solr 中你指定多个 `group.field` 进行分组，并不会嵌套分组查询，而是单独指定的每个域进行分组查询，就好比 SQL 里执行 2 次 Group By：

```
select * from table_name group by location
select * from table_name group by type
```

因此在 Solr Group 中不支持类似 SQL 中的嵌套分组查询，指定多个 `group.field` 进行分组返回的结果集只能是类似下面这样的：

```
-Location X
  -- documents
-Location Y
  -- documents

-Type 1
  -- documents
-Type2
  -- documents
```

关于 Solr Group 的嵌套查询请读者关注 Solr 官方 JIRA，不过在我编写本章时，还是非常遗憾，Solr Group 仍然没有实现嵌套分组查询，至于后续版本是否会实现，请读者访问下面这个链接并关注它的解决状态：

<https://issues.apache.org/jira/browse/SOLR-2553>

## 9.12 本章总结

在本章中，我们主要学习了 Solr Group 的结果集分组，其中按照指定域、任意查询、函数动态计算值等进行分组是本章重点。如果想要更高效的使用 Solr Group，你还需要掌握 Solr 中的 Group 缓存。最后顺带提一提 Solr 中的 Group 分组与 SQL 中的 Group by 之间的区别，明白了使用 Solr Group 可能会有哪些坑，会让你在使用 Solr Group 之前能够心知肚明，提前做好应付的对策。



## Solr 企业级应用

通过第 10 章，你将可以学习到如下内容：

- 如何构建和发布自己的 Solr 版本；
- 如何监控 Solr；
- 在大规模数据量下如何扩展 Solr 索引和查询的负载能力；
- 如何管理你的 Solr Core 以及 Solr 集群。

在前面的章节中，我们都是使用小数据量的索引文档来测试如何使用 Solr 中的核心功能，但有时候，你可能期望能够将 Solr 应用到企业的实际项目中去，使用 Solr 来处理大规模的索引和查询。这也就意味着你在关注 Solr 自身提供的功能之外，还需要额外关注服务器的一些配置，比如 CPU、内存、OS，你还需要关心为了实现需求，需要提供几台服务器，各种服务器之间如何实现数据通信，为了实现 Solr 高负载你需要对 Solr 的哪些配置进行调优，如何监控 Solr 的性能问题，如何调试 Solr 代码，以及当遇到问题时如何去解决它。同时你还需要编写与 Solr Server 进行数据通信的客户端代码实现数据查询（比如使用 SolrJ），还需要考虑如何更高效的将你的数据导入到 Solr 中创建索引。本章将会带着这些问题带领大家学习如何在企业实际项目中使用 Solr。

### 10.1 Solr 源码编译与补丁应用

Solr 官方每年都会随着 Lucene 一起发布多个正式版，但是自从 Solr 完全开源以来，Solr 的主干版本上的代码你可以随时随地获取到，Solr 主干版本上的代码会包含一些实时提交的更新，尽管这些更新可能并没有提交到官方正式发布版本上。主干版本上除了包含一

些实时提交的更新，还包含了大量为解决用户在 JIRA 页面上提交的问题或 BUG 而打的补丁，而这些补丁代码全部来自于全球各地的 Solr 社区开发者的无私奉献，详情请访问 Solr JIRA 页面（<https://issues.apache.org/jira/browse/SOLR>）。大部分补丁会被纳入到官方正式发布版本中，但是官方正式发布版本不会每天都发布新版本，因此你需要学会自己将这些补丁应用到当前使用的 Solr 版本中，以满足你的需求。因为可能你需要的功能官方提供下载的版本并不包含该功能或者官方正式发布版本中有 BUG，但 Solr JIRA 有提供解决此类问题的补丁，那么此时可能需要构建自己的 Solr 发布版本，你可以应用 Solr JIRA 上社区提供的补丁，如果社区没有提供，那么可能需要自己开发修复了。

要开发你自己的 Solr 发布版本，首先需要从 Solr 官网下载你想要使用 Solr 版本的源码以及部署压缩包，至于如何下载请看第一章。当然你也可以通过 Git 来获取 Solr 的源码，具体如下所示：

```
git clone https://github.com/apache/lucene-solr.git lucene-solr
或者
git clone git@github.com:apache/lucene-solr.git
```

如果想将检出的源代码导入到你的 IDE（比如 Eclipse 或 IDEA）中，那么你只需要在源码的根目录（build.xml 所在目录）下执行 `ant eclipse` 或者 `ant idea` 或者 `ant netbeans`，具体请看第一章关于 Solr 源码编译部分。

Solr 源码导入到 IDE 中之后，你有可能需要修改源码并代码调试，最简单的方式就是启用 JVM 的源码远程调试功能，这种方式对于大部分主流的 IDE 来说都适用。

在 IDEA 中调试 Solr 源码步骤如下：

- ❑ 先切到源码根目录下（比如 `D:/solr-5.3.1/solr`）；
- ❑ 然后执行 `ant example`，注意，执行此命令之前，你需要为 ant 添加 ivy 依赖，具体请看第一章 Solr 源码编译部分；
- ❑ 上面命令执行完成之后，切换到 `solr-5.3.1/solr/example` 目录下，执行如下命令：  
`java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=port -jar start.jar`;
- ❑ 这里的 `agentlib:jdwp` 参数会以远程调试模式启动一个 JVM 实例，`address` 参数表示指定一个任意的远程调试端口号，`suspend=y` 参数表示告诉 JVM 等到添加了断点再执行 `start.jar`，如果你想要调试了解 Solr 的启动过程中的具体执行逻辑，那么需要设置 `suspend=y`，否则请设置 `suspend=n`，比如你仅仅只是想调试一个 Solr 查询请求；
- ❑ 然后你需要在 IDEA 中配置远程调试。

如图 10-1 所示单击 Edit Configurations，单击左上角的绿色加号图标，在展开的下拉列表中选择 Remote，然后配置要远程调试的主机域名或 IP 以及端口号 port。

如果使用的是 Eclipse，那么在 Eclipse 中开启远程调试模式如图 10-2 所示，你需要进行如下操作：

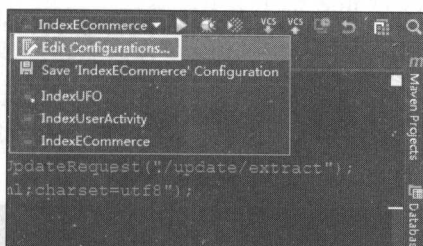


图 10-1 在 IDEA 中打开运行或调试模式的相关配置界面

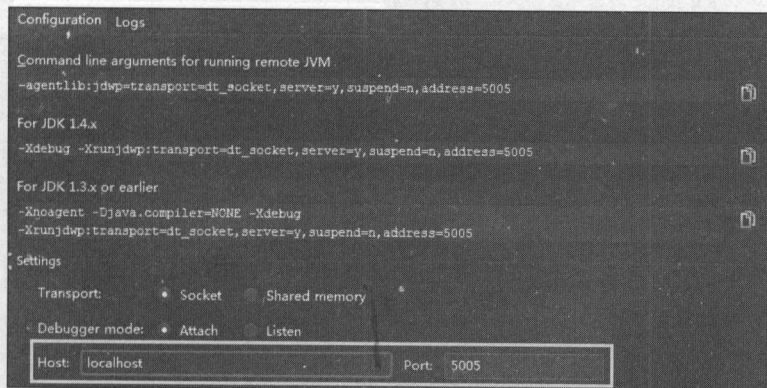


图 10-2 在 IDEA 中配置远程调试模式

Debug As --> Debug Configuration, 然后你会进入 Debug Configurations 界面, 在左侧选择 Remote Java Application 选项进入远程调试设置界面, 然后配置你要远程调试的主机域名或 IP 以及端口号 port。重点是 host 和 port 的设置, host 就是你的服务器 IP 地址或者域名, port 就是你的远程 debug 端口号。



**注意** 远程 debug 必须要保证本地代码和服务端代码一致, 否则调试将失去意义。设置的端口号必须是能够访问的, 不能是其他进程已经占用的端口号。

调试 Solr 源码还有另外一种方式, 那就是直接将 Solr 源码转换成一个 Web Project, 然后直接在你的 IDE 中部署启动它, 就跟调试一个普通的 Web Project 一样。

首先你仍然需要按照第一章介绍的方式对 Solr 源码进行编译, 源码编译完成之后, 需要将其导入到你喜爱的 IDE (IDEA or Eclipse or Netbean) 中, 此时 Solr 的源码还只是一个 Java Project, 你需要将其转换成一个 Java Web Project。先以 IDEA 中为例进行说明:

将 Solr 源码导入到 IDEA 中之后, 请在项目根目录下手动新建 WebContent (或 WebRoot) 文件夹, 然后将 solr.war 包解压后的所有文件复制到 WebContent (或 WebRoot) 文件夹下, 在导航菜单上依次单击 File->Project Structure, 选择左侧的 Facets, 然后单击绿色的加号

图标，在弹出的下拉框列表中选择 Web，然后右侧会显示 Web Project 的配置界面，其中 Deployment Descriptors 用于设置 web.xml 配置文件所在路径，Web Resource Directories 用于配置 Web 资源相关的目录，比如 Web Project 的根路径 "/" 对应的是哪个目录，如图 10-3 所示。

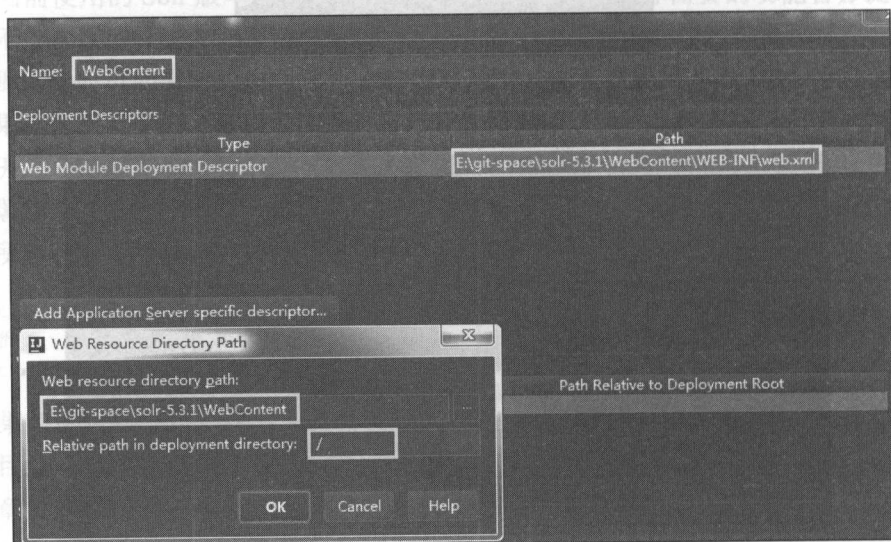


图 10-3 IDEA 中 Web Project 设置

若下方出现黄色的感叹号图标，那么请单击旁边的 create 按钮解决。剩下就是导入一些 Solr 源码依赖的 jar 啦，大部分 jar 包你都可以从 solr 的 zip 压缩包中获取到，然后你将其导入到 WebContent\WEB-INF\lib 目录下并添加到 Classpath。至于到底都依赖哪些 jar 包，这里就不一一说明了，因为依赖的 jar 包太多了不方便说明，具体请读者访问我的 Github 上分享的一个整理好的 Solr 5.3.1 源码：<https://github.com/yida-lxw/solr-5.3.1-web>，我已经将需要依赖的 jar 包进行了很好的分类，请仿照我提供的方式导入所有依赖 jar 包。jar 导入全部完成之后，不出意外代码就不会报语法错误了，此时你就可以像部署普通的 Web Project 一样部署 Solr 并在 IDEA 中启动它了，不过在 IDEA 中启动 Solr 之前，请务必先在 web.xml 中配置 solr.home，如图 10-4 所示。

```

47 <env-entry>
48     <env-entry-name>solr/home</env-entry-name>
49     <env-entry-value>C:/solr_home</env-entry-value>
50     <env-entry-type>java.lang.String</env-entry-type>
51 </env-entry>
52 <!-- Any path (name) registered in solrconfig.xml will be
53 <filter>
54     <filter-name>SolrRequestFilter</filter-name>
55     <filter-class>org.apache.solr.servlet.SolrDispatchFilter

```

图 10-4 web.xml 中配置 solr.home

虽然这种方式比较繁琐，但是一旦在 IDEA 中将 Solr 源码环境搭建好之后，对于以后学习研究 Solr 源码带来了很大便利，你可以在 IDEA 中随心所欲的打打断点进行调试，对你了解 Solr 内部执行逻辑很有帮助哦！

如果你平时习惯使用 Eclipse，那么你同样也可以将 Solr 源码导入到 Eclipse 中并实现在 Eclipse 中进行部署启动。首先你同样需要将 Java Project 转换成 Web Project，具体操作步骤如下：

修改 Solr 源码根目录下的 .project 文件，如下所示：

```
<natures>
  <nature>org.eclipse.jdt.core.javanature</nature>
  <nature>org.eclipse.wst.common.project.facet.core.nature</nature>
  <nature>org.eclipse.wst.common.modulecore.ModuleCoreNature</nature>
  <nature>org.eclipse.jem.workbench.JavaEMFNature</nature>
</natures>
```

在 Eclipse 中项目上鼠标右键→properties，然后选择“Project Facets”，选择 java、Dynamic Web module。

然后修改项目根目录下 .settings 文件夹中的 org.eclipse.wst.common.project.facet.core.xml 文件，将其中的 jst.web 值由 3.0 修改为 2.5，如下所示：

```
<?xmlversion="1.0"encoding="UTF-8"?>
<faceted-project>
  <fixedfacet="wst.jsdt.web"/>
  <installedfacet="java"version="1.6"/>
  <installed facet="jst.web"version="2.5"/>
  <installedfacet="wst.jsdt.web"version="1.0"/>
```

然后你需要将 solr.war 包解压后的所有文件复制到 WebContent（或 WebRoot）下，剩下的步骤跟 IDEA 中类似，就不赘述了。

有可能你在使用 Solr 的过程中意外发现了 Solr 的一个 BUG 或者你需要的某个功能暂时 Solr 并不支持，因为 Solr 是一个开源项目，并且有大量的社区开发人员予以支持，可能其他开发人员也遇到此类问题，而他为此打了个补丁解决了此类问题，你可以通过访问 Solr 官方 JIRA 了解详情：<https://issues.apache.org/jira/browse/SOLR>。

用户可以在 Solr 的 JIRA 上提交问题或 BUG，或者上传自己为了解决某个 BUG 或者实现某个 Solr 尚未支持的功能而做的代码更新，一般提交的代码更新会以 Patch File（补丁文件）的形式进行上传，你可以在上面寻找你正遇到的问题，可能会提交的补丁又有可能一直是未解决状态。

如果有提供补丁文件，你可以应用补丁文件以解决你的问题。在下载补丁文件时，你应该选择离当前时间最近的那个补丁文件，同时需要注意对方上传的补丁文件针对的版本号，如果你使用的 Solr 版本与补丁文件不一致，那么你应用补丁文件后可能会抛出异常或者没有效果。Solr 社区使用的补丁文件格式是使用 svn diff 命令或者 git format-patch 命名创



建的。

测试一个补丁文件是否适用于你当前的 Solr 版本，你可以执行如下命令进行测试：

对于是使用 `svn diff` 命令创建的补丁文件，你需要执行如下命令：

```
wget <URL to the patch> -O - | patch -p0 --dry-run
```

<URL to the patch> 表示补丁文件的下载地址，即通过 Linux 下的 `wget` 工具去自动帮你下载补丁文件。如果提供的补丁文件是使用 `git format-patch` 命令创建的，那么你需要执行如下命令进行测试：

```
wget <URL to the patch> -O - | patch -p1 --dry-run
```

没错，这里的 `-p0 -p1` 选项即表示不同的补丁文件格式。上面的两种方式都是借助 Linux 系统的 `wget` 工具去自动帮你下载补丁文件，当然你也可以自己手动去下载补丁文件到本地硬盘，然后通过如下命令进行测试：

```
patch -p0 -i <patchfile> --dry-run
```

或

```
patch -p1 -i <patchfile> --dry-run
```

这里的 <patchfile> 参数即表示你的补丁文件存放路径。上面的这些命令都包含了一个 `--dry-run` 选项，它表示仅仅只是测试当前补丁文件适用于当前 Solr 版本，并不是实际去修改源码应用补丁文件，在应用一个补丁文件先进行测试这是一个很好的实践方式。如果你想正式应用某个补丁文件，请移除 `--dry-run` 选项。如果你在应用补丁文件过程中操作有误，你想撤回，那么可以执行如下命令实现：

```
svn revert -R ./ // svn 下
```

或者

```
git reset --hard HEAD // git 下
```

了解了如何导入 Solr 源码到 IDE 中，以及如何应用 JIRA 上提供的补丁文件，那么你就可以随心所欲的编译打包部署属于你自己的 Solr 发布版本啦！Solr 的 `build.xml` 提供了很多自定义的 Ant Task，你只需要在 IDE 中双击相应的 Ant Task 即可完成 Solr 源码的编译与打包，从而形成属于你自己的发布版本。

## 10.2 部署 Solr

Solr 提供了一个 war 包文件，你可以直接将其部署到任意的 Servlet 容器中，如果你不清楚如何将 war 包文件部署到 Servlet 容器中，请搜索相关知识进行了解。当你使用 Solr 提供的 `start.jar` 方式来运行 Solr 时，其实本质是启动了内嵌的 Jetty Servlet 容器。本小节将包含如下内容：如何构建你自己的 Solr 发布版本以及如何在企业产品环境中进行部署。



### 10.2.1 构建你自己的 Solr 发布版本

当你下载了 Solr 官方提供的发布版本压缩包文件，在解压后的 X:/solr-5.3.1/server/solr-webapp/webapp 目录下你会看到有一个 solr.war 包文件，如果发现没有提供 war 包，那么你可以手动打个 war 包，如下所示：

```
E:\solr-5.3.1\server\solr-webapp\webapp>jar -cvf solr.war ./*
```

### 10.2.2 Embedded Solr

除了可以将 solr.war 包部署到 Servlet 容器，Solr 还支持将 Solr 嵌入到其他 Java Project 中通过 Solr API 方式来直接访问，而不是通过 Http 请求方式来访问交互即 Embedded Solr。当你想要为你的 Java Project 添加搜索功能，但是又想要 Solr 能够包含在你当前项目中且只有当前项目能够访问 Solr，而不是暴露成一个搜索服务，其他项目也能访问到，这个时候采用嵌入式 Solr 可能会比较适合你。你可以通过 SolrJ 来访问嵌入式 Solr 服务。不管是考虑将 Solr 嵌入到当前项目中还是单独部署到一台服务器上，你都需要确认你是否有合适的服务器硬件设备以及你的操作系统是否已经正确配置了，这都关乎 Solr 的执行性能。

## 10.3 Solr 硬件要求与系统配置

Solr 支持对数十亿的索引文档进行查询秒级返回，但是单机的处理能力总是有瓶颈的，对于大文本的索引文档，每个 Solr Server 只能存储大概几百万的索引文档，但是对于大量小索引文档，Solr 可能能够存储几百亿甚至更多还能保证稳定的查询速度。尽管没有一个固定的最优配置来适用于任何使用场景，但是还是有一些通用的原则能给予你一些指导方向。

### 10.3.1 内存和 SSD

如果你拥有自己的服务器（不是阿里云租来的云服务器），为了保证 Solr 的运行性能，你需要为你的服务器添加足够大的内存，内存相对于服务器的其他硬件来说价格还算比较便宜，而 Solr 刚好又需要占用大量内存。但是如果你使用的是云服务器，内存每上一个等级价格都越来越昂贵，这时你需要考虑下金钱成本。

在 Solr 的核心操作中，Solr 缓存是最消耗内存的，比如 Facet、Sort、索引创建等这些操作都需要使用缓存。你必须确保你有足够的内存来支撑这些操作。你可以在 Solr 的 Web 后台界面的主页上看到当前服务器的所有内存大小以及已经占用了多少。此外你还需要确保你的服务器剩余的未分配给 JVM 的可用内存是否足够在查询时缓存索引数据，假如你的索引数据大小比你服务器的剩余可用内存还大，这也就意味着 Solr 会尝试一定的磁盘 IO 来保证查询的吞吐量。

一些用户为了加快磁盘寻址速度，因而选择为服务器购买 SSD（固态硬盘），但是你没

有必要一开始就直接上 SSD。如果能保证你的所有索引数据一直都缓存在内存中且不会溢写到硬盘,那么 Solr 查询性能可能会是最高的。如果你需要的是更高的索引吞吐量,这时候采用 SSD 硬盘才会极大的发挥 SSD 硬盘的优势,因为创建索引就是一个磁盘 IO 极其密集的操作。在 Solr 查询的时候使用 SSD 固然能缓解一定的磁盘溢写带来的性能损耗,但是并不能充分发挥 SSD 的优势,就好比买了本书,那本书非常好,但是你 1 年都看不了 10 页,那你买来了也是一种浪费,并不是说那本书不好。在大部分操作系统中,一旦一个文件被加载,那么它就会一直存在于内存中,除非操作系统内存强制回收。操作系统会缓存那些最近加载的文件,直到系统的所有内存快消耗完时,操作系统会尝试将哪些最近访问最少的文件所占用的内存进行释放回收以分配给新数据使用。

因为 Solr 需要多次访问硬盘上的索引数据,因此为了提升文件读取性能,需要将索引数据缓存在内存中。尽管索引数据体积可能会有几百 G,甚至超过了一台服务器的剩余内存所能缓存的最大容量,但是只要你能确保你的索引体积一直小于你的操作系统的总内存大小,而且你的索引数据已经全部缓存到内存中了,那么你在执行 Solr 查询时就不会发生索引数据溢写操作。在大多数情况下, Solr 查询其实都是基于内存的查询,这使得你的索引创建和 Solr 查询变得更快。因此需要尽量保证操作都是在内存中进行,尽量减少磁盘 IO。

### 10.3.2 JVM 配置

这里我们并不会讨论 JVM 垃圾回收机制以及 JVM 如何调优,我们只关心那些会影响 Solr 执行性能的 JVM 配置参数。因此首先需要熟悉一些 JVM 配置参数的含义:

- ❑ -Xms: 初始堆大小;
- ❑ -Xmx: 最大堆大小;
- ❑ -Xmn: 新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。  
实际可用空间 = Eden + 1 个 Survivor, 即 90%;
- ❑ -XX: NewSize = n: 设置年轻代大小;
- ❑ -XX: NewRatio = n: 设置年老代和年轻代的比值。如果值为 3, 表示年轻代与年老代比值为 1: 3, 年轻代占整个年轻代年老代和的 1/4;
- ❑ -XX: SurvivorRatio = n: 年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如果值为 3, 表示 Eden: Survivor = 3: 2, 一个 Survivor 区占整个年轻代的 1/5;
- ❑ -XX: PermSize = n: 永久代(方法区)的初始大小;
- ❑ -XX: MaxPermSize = n: 设置永久代大小;
- ❑ -Xss: 设置栈容量; 对于 HotSpot 来说, 虽然 -Xoss 参数(设置本地方法栈大小)存在, 但实际是无效的, 因为在 HotSpot 中并不区分虚拟机和本地方法栈;
- ❑ -XX: PretenureSizeThreshold: (该设置只对 Serial 和 ParNew 收集器生效) 可以设置进入老生代的大小限制;
- ❑ -XX: MaxTenuringThreshold = 1: (默认值, 最大值都是 15) 表示一个对象在年轻

代经过多少次 GC 后会被移动到年老代，如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概率。对象的 age 是由 4 个 bit 来表示的，这就意味着，对象的 age 最大值就是 15。如果我们将 MaxTenuringThreshold 设置的超过了 15，对象的 age 将永远也达不到；

- ❑ -XX: TargetSurvivorRatio: 一个计算期望存活大小 Desired survivor size 的参数；
- ❑ 计算公式:  $(\text{survivor\_capacity} * \text{TargetSurvivorRatio}) / 100 * \text{sizeof}(\text{a pointer})$ : survivor\_capacity (一个 survivor space 的大小) 乘以 TargetSurvivorRatio, 表明所有 age 的 survivor space 对象的大小如果超过 Desired survivor size, 则重新计算 threshold, 以 age 和 MaxTenuringThreshold 的最小值为准, 否则以 MaxTenuringThreshold 为准。但对于开启了 UseAdaptiveSizePolicy 的 ParNew GC 而言, XX: TargetSurvivorRatio 配置是也无效的, 默认是不开启的, 在设置 -XX: -UseAdaptiveSizePolicy (注意是减号) 后, 则以 MaxTenuringThrehsold 为准, 并且不会重新计算 threhsold;
- ❑ -XX: + UseParNewGC: 对 Yong 区域启用并行回收算法;
- ❑ -XX: + UseParallelGC: 一种较老的并行回收算法;
- ❑ -XX: + UseParallelOldGC: 对 Tenured 区域使用并行回收算法;
- ❑ -XX: ParallelGCThread = 10 : 并行的个数, 一般和 CPU 个数相对应。年轻代的并行收集线程数计算公式默认是  $(\text{ncpus} \leq 8) ? \text{ncpus} : 3 + ((\text{ncpus} * 5) / 8)$ ;
- ❑ -XX: + UseAdaptiveSizepolcley: 收集器自动根据实际情况进行一些比例以及回收算法调整;
- ❑ -XX: MinHeapFreeRatio: 表示剩余空间百分比多少时, 开始减小 committed 的内存;
- ❑ -XX: MaxHeapFreeRatio : 表示剩余空间百分比多少时, 开始增加 committed 的内存, 直到 -Xmx 大小;
- ❑ -XX: MaxGCPauseMillis 指 GC 最大的暂停时间, 当超过这个时间, 那么 JVM 会适当调整内存比例 (前提是使用的是基于比例的 YONG 和设置);
- ❑ -XX: + UseConcMarkSweepGC: 启用 CMS 回收器, 一般针对 Tenured 区域 (即老年代);
- ❑ -XX: CMSFullGCsBeforeCompaction = 3: 多少次 GC 后会进行压缩碎片;
- ❑ -XX: + UseCmsFullCompactAtFullCollction : 打开老年代压缩 (所谓压缩其实就是把内存拷贝到连续的空间, 减少内存碎片, 相当于 windows 里常用的磁盘碎片整理);
- ❑ -XX: + CMSIncrementalMode: 增量 GC, 将内存切块, 分布在多个局部去 GC;
- ❑ -XX: CMSInitiatingOccupancyFraction : 在并发 GC 下, 由于一边申请内存, 一边 GC, 就不能在不够用的时候 GC, 默认情况下是在使用了 68% 的时候进行 GC, 通过该参数可以调整实际的值。这里如果设置为 65 即表示当内存使用率达到了 65% 就触发老年代的 Full GC。对于高并发的程序, 建议这个值设置为 60 ~ 80 之间, 具体值自己去压测。

如果你追求的是系统吞吐量，尽可能的减少系统执行垃圾回收的总时间，那么你可以使用新生代并行垃圾回收器即 `-XX: + UseParallelGC`。

如果你追求的是系统低停顿，尽量减少 FullGC 的次数，尽量将对象预留在年轻代，因为年轻代的 MinorGC 执行开销要远小于年老代的 FullGC，此时你可以使用 CMS 垃圾回收器即 `-XX: + UseConcMarkSweepGC`。关于 JVM 垃圾回收器更多知识请读者阅读《深入 Java 虚拟机》这本书。

基于以上对 JVM 参数的了解，以及 Solr 的性能极其依赖于内存使用情况，因此我们需要为运行 Solr 的 JVM 实例提前分配足够多的内存，如下所示：

```
java -Xms2g -Xmx2g -jar start.jar
```

或者

```
java -Xms2048m -Xmx2048m -jar start.jar
```

如果你的 Solr 是部署在 Tomcat 中，那么你需要修改 Tomcat 的 `catalina.bat`，在文件的开头添加如下内容：

```
set JAVA_OPTS=-Xms2g -Xmx2g
```

对于 Linux 用户，那么你需要修改 Tomcat 的 `catalina.sh` 文件，在文件的开头添加如下内容：

```
JAVA_OPTS=-Xms2g -Xmx2g
```

这里我们给运行 Solr 的 JVM 实例分配了 2G 内存，这里设置 2G 只是一个示例，并不意味着 2G 就一定够用，你的 Solr 实例可能只需要 1G 内存就够用，又或许需要 10G 呢？但是在你提前为 Solr 实例分配内存之前，必须确保你分配的内存足够用，不会减缓 Solr 处理速度，正如前面讨论的那样，Solr 是极度消耗内存的，你需要确保 Solr 的核心数据结构（比如 Solr Core、Solr 缓存、Solr 的其他内存结构）能够缓存在内存中从而能够执行查询操作。如果提前分配过大的内存给运行 Solr 的 JVM 实例，那么垃圾回收时间会增多，因为你需要回收更多的垃圾，同时操作系统就没有足够的剩余内存去缓存 Solr 索引，那么此时磁盘 IO 读写操作自然就增多。一般的设置经验是稍微给 Solr 多一点内存缓冲，剩余的留给操作系统。因此你的操作系统拥有足够的剩余内存去缓存 Solr 索引文件极其重要。

Java 中垃圾回收是个令人头痛的问题，人们经常会问 Solr 中应该使用哪种垃圾回收器，这完全取决于你的系统需求。正如我前面所说的，如果你追求的是 Solr 大部分时间拥有最佳性能，但是可以容忍偶尔长时间的由于垃圾回收导致的系统卡顿（系统卡顿期间，Solr 会响应迟钝或者无响应），那么吞吐量垃圾回收器会比较适合你即 `ParallelGC`，这也是 64 位操作系统默认开启的垃圾回收器。如果你更多追求的是性能的一致性即你能容忍查询平均性能慢点但系统卡顿次数尽量少且每次卡顿持续时间尽量短，那么 CMS 垃圾回收器会比较适合你。

```
java -server
```

```

-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
-XX:CMSInitiatingOccupancyFraction=80
-Xmx$JAVA_HEAP_SIZE
-Xms$JAVA_HEAP_SIZE
-jar start.jar

```

如果你的 Solr 是部署在 Tomcat 中，那么需要将其配置在 `catalina.bat` 或 `catalina.sh` 文件中。上面的配置对于大部分用户来说够用了，但是你还是需要查阅 JVM 相关文档中关于垃圾回收的配置参数部分，根据上面的配置示例在你的 Solr Server 上进行性能测试。很多垃圾回收器提供的一些可选参数能够优化垃圾回收性能，但是垃圾回收器性能调优超出了本书的范畴。如果你发现当前使用的垃圾回收器严重影响了 Solr Server 性能，你可以网上查阅有关 JVM 内存调优相关的在线资料，可能会帮助到你。

### 10.3.3 思考 Solr 索引与查询性能

前面我们讨论了内存和操作系统在内存中缓存最近常访问的 Solr 索引文件的能力的重要性。那么当我们往 Solr 中添加一个新的索引文档或者更新了一个索引文档时会发生什么呢？索引重建过程对 Solr 性能会产生重大影响。这是一个值得深入了解的方面。

Solr 内部是借助 Lucene 来创建倒排索引表的，索引创建永远都是一个增加索引的过程，当索引文档更新了只会添加一个新的索引文档，删除之前的旧索引文档。理论上讲，这意味着一旦一个索引文件被写入，它就不能被更新，同时也意味着为了实现在服务器之间的索引数据复制和更新，只有那些包含了增量更新部分的索引文档会被复制。因为索引数据体积经常是几十甚至几百 GB，索引采用这种增量性质在很多方面会带来好处：

- ❑ 硬盘占用需求降低了，因为之前的索引文件会以多个新的增量版本被共享；
- ❑ 在 Server 之间索引复制需要传输的只是增量索引文件；
- ❑ 不论什么时候提交一个新的索引文档，操作系统的文件系统内存中已经缓存了大部分的索引文件。

索引增量处理需要一些执行开销，主要还是由于之前的旧索引文档永远不会被更新，任何一个索引文档的更新或删除操作实际都需要额外的磁盘空间，因为它们都是写入一个新的索引文件，因此，如果你将同一个索引文档提交多次，你会发现你的索引体积在不断增长，因为旧版本的索引文档和新版本的索引文档全都在索引中。当删除一个索引文档时，内部会维护一个黑名单用来告诉 Solr 忽略哪些旧版本的索引文档，那些旧版本的索引文档仍然驻留在早已存在的索引段文件中。除非你显式的执行一个 Solr 硬提交，才会真正的删除一个索引文档。默认索引的添加和删除操作都是类似收集代办任务的过程，并没有真正立即去执行，而是累积到一定量触发了某个提交点才会真正去执行索引写入操作。当我们说到旧索引文件时，需要澄清说明的是，有时候在很多索引操作提交之后，很多段文件需要合并，段文件合并操作并不是修改之前的段文件，而是创建一个新的段文件，然后弃用之前的段文件。当段文件合并完成之后，旧段文件才会被删除，这也就意味着在段文件合并过程中，你



的段文件硬盘占用是双倍的。

要使一个新的段文件中的索引文档被查询到，你需要发起一次索引提交，Solr 的 Hard commit（硬提交）会为新的增量段文件和旧的段文件开启一个新的 IndexSearcher，旧的 IndexSearcher 用于在新的 IndexSearcher “上任”之后站好最后一班岗。

当一个硬提交被发起，Solr 会随即创建一个新的 IndexSearcher 实例，它会加载仍然存在于索引中的旧段文件以及新添加的增量段文件。因为 Solr 需要持续接收查询请求，为了保证任何时刻一个新的索引文档被添加进来，都能立即被查询到，开启一个新的 IndexSearcher 实例是有很重要意义的。

Solr 通过并行的持有两个 IndexSearcher 实例，实现在旧版本的索引与新版本的索引之间的无缝转换过渡。当新的 IndexSearcher 实例尚未加载完成之前，旧的 IndexSearcher 实例会继续接收查询请求，因为 Solr 需要会为之之前的普通查询、Filter Query、Field 值以及其他存在于内存中的数据使用缓存，用以提升查询执行速度。在新的 IndexSearcher 实例加载完成可以接任旧 IndexSearcher 实例接收查询请求之前，那些缓存仍然可以被使用。不过遗憾的是，因为那些缓存都会被绑定在一个指定的索引版本上，这也就意味着旧 IndexSearcher 实例的缓存不能应用于新的 IndexSearcher 实例上，你需要提前将旧 IndexSearcher 实例的缓存复制到新的 IndexSearcher 实例上，这个过程叫做“Auto Warm”即自动预热过程。主要是为了保证新的 IndexSearcher 实例加载完成之后的查询依然能直接命中缓存而不是去硬盘加载索引文件，最终目的还是为了 IndexSearcher 实例启动初期查询性能依然能杠杠滴。说的更直白点就是每次当一个硬提交被发起，会随即创建一个新的 IndexSearcher 实例，同时会伴随着为新的 IndexSearcher 实例进行缓存“自动预热”操作。

新的 IndexSearcher 实例需要从旧的 IndexSearcher 实例中“自动预热”多少缓存以及“自动预热”多少个缓存对象，这些你都可以在 solrconfig.xml 中进行配置，示例如下所示：

```
<fieldValueCache class="solr.FastLRUCache" size="500" initialSize="50"
autowarmCount="500" />
<filterCache class="solr.FastLRUCache" size="500" initialSize="500"
autowarmCount="250" />
<queryResultCache class="solr.LRUCache" size="100" initialSize="100"
autowarmCount="0" />
<documentCache class="solr.LRUCache" size="500" initialSize="500"
autowarmCount="0" />
```

在某些时候，你可能希望配置更大的缓存，尤其是当你已经为 JVM 实例分配了足够大的内存时，当你增大缓存大小，查询的缓存命中率也会呈线性增长。那么任何时刻新的 IndexSearcher 实例启动时需要自动预热的缓存对象个数也随之增多。当然如果有足够的内存予以支持，那么缓存设置大些是可以的，但是随之带来的自动预热需要复制的对象增多，也就意味着自动预热时间加长了，即 IndexSearcher 实例启动完成时间加长了，那么实时提交的新索引文档能够被立即查询到的及时性就差了，也就是说你的 Solr 查询实时性就没法保证了，尤其是当你有大量执行开销昂贵的 Filter Query，你的 IndexSearcher 实例自动预热



过程可能需要花费几分钟甚至几个小时，这对于实际的搜索项目而言，显然很不现实。然而，如果你将自动预热的对象个数设置的过小，那么你的 `IndexSearcher` 实例的启动完成时间是加快了，但是初期的查询性能会受影响，比如查询耗时变大，甚至会出现响应卡顿，因为此时的查询可能无法命中缓存了。如果出现此类情况，那么此时你需要调整缓存对象的个数已达到一个合理的自动预热配置，既能保证 `IndexSearcher` 实例能高效的及时预热又能兼顾自动预热完成之后的查询性能。

当索引段文件变得很大，Solr 会自动触发段文件合并去压缩索引，所谓段文件合并其实就是删除已经标记为需要删除的索引文档以及版本重复的索引文档。段文件的合并时机是由 `Merge Policy` 策略类决定的。`Merge Scheduler` 和 `Merge Policy` 共同决定了段文件如何合并，你可以在 `solrconfig.xml` 文件中对其进行配置：

```
<mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler">
  <int name="maxMergeCount">4</int>
  <int name="maxThreadCount">4</int>
</mergeScheduler>
<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
  <int name="maxMergeAtOnce">10</int>
  <int name="segmentsPerTier">10</int>
</mergePolicy>
<mergeFactor>10</mergeFactor>
```

`TieredMergePolicy` 是 Solr 中的默认段文件合并策略，它通常是最高效的。`maxMergeAtOnce` 和 `segmentsPerTier` 这两个参数决定了 Solr 段文件什么时候被合并。`segmentsPerTier` 参数决定了在 Solr 开始合并段文件之前可以创建的段文件个数。`maxMergeAtOnce` 参数决定了一次段文件合并操作中可以合并到新的段文件中的段文件个数。

不过 Solr 中还有几种索引合并策略：`LogByteSizeMergePolicy`、`LogDocMergePolicy`、`AlcoholicMergePolicy`，一般也不需要显式的切换索引合并策略，保持默认的 `TieredMergePolicy` 即可。

通常，将合并因子设置得越大，索引创建速度会越快。段文件个数越大，你的查询速度就越慢，因为查询需要加载遍历更多的索引文件，而且每个索引文件可能包含了旧版本的索引文档或者已经标记为需要删除。

段文件的合并相关参数该如何设置更合理，这取决于你的实际需求，是更关心索引创建更快速，还是查询速度更快速呢？所以怎么设置你懂的。

有时候，可能需要将所有的段文件合并成一个段文件进行索引优化，如果你想要在索引创建完成之后执行一次索引优化，将索引段文件全部合并成为单个段文件，以提升查询速度。此外，如果你需要执行大量的索引更新操作（比如每天更新索引数据），那么周期性地执行所有索引优化释放索引中的旧索引文档占用的磁盘空间将会非常有意义。你可以传递 `<optimize/>` 标签给 `/update` 请求处理器来请求对索引进行优化，示例如下所示：

```
http://localhost:8080/solr/core1/update?stream.body=<optimize/>
```

不过需要谨记的是索引优化是一个执行开销很昂贵的操作，它会逐行重写整个索引，可能会花费很长时间以及很多系统资源。当时如果你的索引合并很频繁，那么索引优化操作可能会很快，因为在索引优化之前，段文件已经被重写了。但是你仍然避免不了新旧索引所在段文件同时在系统缓存中存在，双倍的系统内存占用可能会影响你的查询性能，当你的系统内存不够用的时候可能表现得会更加明显，你此时可以通过将索引文件拆分到同一台服务器的多个 Core 上对索引优化操作可能遇到的内存双倍占用问题进行优化，因为索引优化一次只会针对一个 Core 进行。

Solr 的代码是基于 Java 编写的，这也就意味着 Solr 可以运行于各种系统平台之上，但是大部分的 Solr 集群还是选择运行于 Linux 操作系统。本章节我们来交流下一些关于能够提高运行于 Linux 上的系统性能的有效技巧。

正如前面讨论过的那样，为 Solr Server 提供足够的内存（一部分用于缓存你的整个索引数据，一部分分配给运行 Solr 的 JVM 实例）能够极大地提升 Solr 的执行性能，这样操作系统才能从硬盘上加载索引文件并缓存到系统内存中。然而遗憾的是，Solr 初始化的时候，需要加载所有索引文件，这意味着系统重启之后你第一次启动 Solr 或者系统中有其他文件被加载导致 Solr 的部分索引文件被从系统缓存中“踢出去”，那么此时 Solr 运行会比平时要慢很多。所以在 Solr 启动时你可以指定某些加载很缓慢的索引文件进行延迟加载，从而加快 Solr 的启动速度。你可以通过执行下面的命令在 Solr 未启动之前提前加载 Solr 索引文件到内存中：

```
find /opt/solr_home/ecommerce/data/ -type f -exec cat {} \; > /dev/null
```

假定你已经使用了默认的目录结构来存储你的每个 Core 的 Solr 索引，比如上面的 /opt/solr\_home/ecommerce/data/。这个命令会从硬盘上加载指定的索引数据目录下的所有文件到系统内存中。对于大数据量的索引数据（比如几十或几百 GB），那么提前加载索引文件到系统内存中可能需要花费几十分钟甚至更长时间，但是 Solr 启动之后的查询速度变快了，尤其是当你刚刚重启了系统服务器之后效果更明显。

演示这种方式是否会有效果的最好方式就是运行这个命令两次，一次是在系统服务器刚刚重启之后，另一次是在 Solr 第一次运行之后，你会发现第二次执行该命令很快，几乎是瞬间的，因为第二次加载文件的时候文件已经存在于内存中了。在 Solr 启动之前提前加载 Solr 的索引文件是一个非常有用的小技巧。假如你的索引文件体积比你的系统剩余内存还要大，那么在执行索引文件提前加载命令时，就不是所有文件都会被加载到内存中。但是需要注意的是，一定要是在 Solr 启动之前执行该命令，如果你的 Solr 实例已经在运行，此时索引文件应该已经被加载到内存中了，再执行索引文件加载就没有太大意义了。

由于 Lucene 采用增量索引的处理机制，一个索引文件可能由几百个文件组成，而这些所有文件需要同时打开。如果你的 Solr 索引中包含了大量的索引文件，这意味着 Solr 为了支持查询功能，需要在任意时刻同时打开几千个文件，遗憾的是，对于基于 Unix 的操作系

统默认支持的同时打开文件个数是有上限的，如果达到了这个上限值，那么会导致 Solr 崩溃。不过你可以通过 `ulimit-n` 命令来取消这种限制，此命令表示可以打开的文件描述符总个数，为了增大这个限制保证 Solr 能正常安全的运行，你可以如下进行指定：

```
ulimit -n 100000
```

很多操作系统的默认文件描述符最大值是 1024，但是这个默认值对于大部分的 Solr 应用来说都是不够用的，所以你有必要适当调整这个文件描述符限制，尤其是当你期望在你的 Solr Server 上运行多个 Solr Core 时会非常有用。应用 `ulimit-n` 命令仅仅只是作用于当前程序会话内有效，重启系统之后还是默认值，如果想要永久性进行设置，那么你需要进行 `vi` 编辑 `/etc/security/limits.conf`，你需要确保你设置的文件描述符足够大，否则一旦又达到了上限值，那么你的 Solr 又会崩溃了。

## 10.4 Solr 数据批量导入

到目前为止，你已经知道了如何提交一个索引文档到 Solr Server 进行索引创建，你可以通过 `Http` 请求 `/update` 这个请求处理器来完成索引的提交。你可以借助 `post.jar` 或者通过 Solr 提供的 Web 管理后台界面，甚至你还可以使用 SolrJ 客户端 API 来实现。Solr 允许你发送不同格式（比如 XML/JSON/CSV）的数据给 `/update` 请求处理器，同时也可以使用 SolrJ 来帮助你提交各种格式的数据文件到 Solr，它可以帮助减少额外的数据序列化和反序列工作，简化了你的开发工作量。当你提交一个索引文件给 Solr 时，索引文档所需要花费的时间由很多因素决定，一部分是由 Solr 内部的配置决定的，比如我们在前面讨论过的索引段文件合并与索引优化相关配置。但是索引文档的输入格式、网络连接数和网络延迟、从原始数据源拉取数据的速度、请求 Solr 的线程数，这些所有因素影响着最终的响应时间。对于网络延迟可能无能为力，但是你可以采用批量提交的方式来减少网络间的这种往返请求次数。比如采用如下格式一次性提交多个索引文档，而不是每个索引文档提交一次：

```
<add>
<doc>
...
</doc>
<doc>
...
</doc>
...
</add>
```

采用批量提交索引文档的方式可以显著提高你的索引吞吐量，因为你减少了网络之间的连接次数并且增大了 Solr 一次请求的工作量。通常一次请求发送几百个大文本索引文档是合理的。如果是小文本索引文档，那么你一次性可以批量发送几千个。具体一次批量发送

多少个，这个需要你自己去进行大量实验，最后根据你实际一次批量提交的索引文档个数和每次请求的耗时来得出适合你当前项目的具体最优值。

除了批量提交能影响索引吞吐量，对不同格式的文件进行索引同样也会影响索引吞吐量。解析 XML 需要大量的根据 XPATH 表达式进行数据解析工作，而且 XML 格式数据非常冗余，更占内存且网络传输量也更有压力，这也就意味着一般选择其他的文档输入格式能提升索引性能，比如选择 javabin、或 CSV 格式。但 Solr 的索引处理能力终究还是有瓶颈的，所以你需要监控你的 Solr Server 的 CPU 和内存等系统资源使用情况，从而实现在 Solr 索引和查询变慢之前提前预判 Solr 的运行性能，如果确定系统资源快不够用了，那么你最好是将你的索引数据分成多个 Core 跨多服务器存放，并且跨服务器来分摊你的索引 update 请求（比如使用 SolrCloud 来管理这些操作）。

## 10.5 Solr Shard 与 Replication

Solr 允许你创建多个查询索引，每个索引由一个 Solr Core 表示。你可以将你的索引数据进行分片（或者说分区），每个分片中的数据可能会跨多个 Core，这就叫做“Solr Shard”。复制某个分片的索引数据作为一个副本，这就叫做“Solr Replication”。关于如何对索引数据进行分片，如何创建索引副本，以及如何使用 SolrCloud 实现分布式查询，我们会留到 SolrCloud 章节做详细讲解。本章节我们主要讨论如何确定你的 Solr 集群的 shard 和 replication 的个数。

### 10.5.1 Shard

当你单个服务器需要处理大量的索引文档时，那么对索引数据进行分片然后再处理会很有意义。图 10-5 形象得演示了什么是 Solr Shard，以及如何简单的实现对 Solr 索引进行分片。

当你搭建了一个 Solr 集群，你需要考虑将索引数据分割成多少个 Shard，你可以设置为一个 Sahrd 或者几百个 Shard，这取决于你的索引体积。如果你每个索引集合只有一个 Shard，那么你的索引就不算是分片了。本章会提供一些指导来帮助你决定需要为索引数据划分多少个 Shard。但是在做决定得出最优值之前，你需要考虑很多因素以及对数据进行大量测试。

Solr 索引数据分片并不能解决自动故障容错问题，索引分片能有效缓解你的索引数据不断增长带来的索引和查询压力。在开始决定索引分片数量之前，你应该考虑以下 5 个因素：

- 索引文档的总数量；
- 每个文档的大小；
- 要求的索引吞吐量；
- 查询的复杂度；
- 预期索引的增长量。

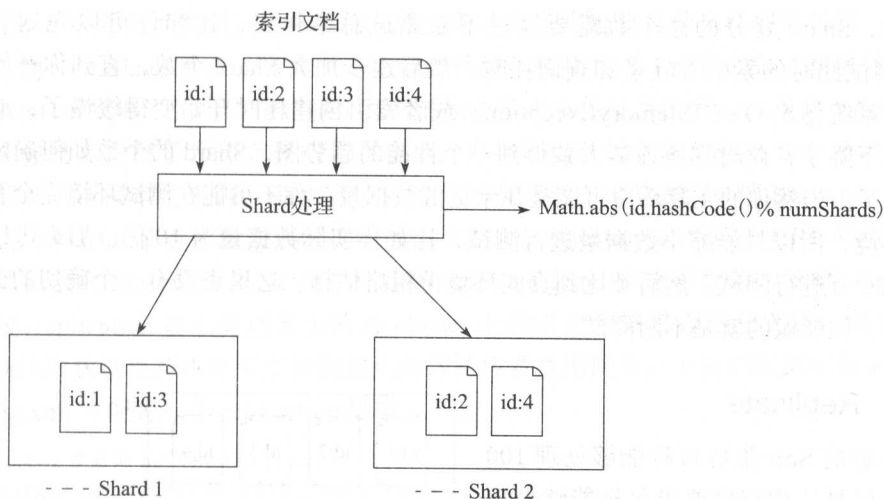


图 10-5 形象演示何为索引 Shard (分片)

索引文档总数量很大时，你需要测试出每个节点最多能处理多少索引文档，然后计算出 Shard 的个数。一般来说，索引文档的总数越大，你需要划分的 Shard 个数也应该越大。

当然，如果你的实际索引文档都是一些大文本的索引文档，那么你测试的时候也应该尽量去模拟真实环境，如果测试的时候采用一些小文本的索引文档进行测试，可能会估算不准确。此外一些大文本的索引数据一般意味着需要更多的内存和磁盘空间来存储，因此你的 Shard 个数也应该考虑相应的增大。

如果你要求高索引吞吐量，那么你应该尽量充分利用每个节点的资源，使得每个节点的吞吐量都达到饱和，那么整个索引吞吐量才是最佳的。如果你追求的是索引时间尽量快速，那么应该考虑的是每个 Shard 需要处理的索引文档数量应该尽量少，因此这个时候 Shard 个数应该规划大一些，反之，如果你更多考虑的是索引吞吐量，那自然是要求每个 Shard 节点尽量多的处理索引文档而不能让它长时间处于空闲状态。此时 Shard 个数可以相对小些。当然你还要同时兼顾查询的性能。

如果你的查询是很复杂的，那么意味着它需要更多的系统执行资源，如果 Shard 个数分配的过小，势必意味着单个 Shard 上要处理的文档增多，查询执行开销也会增大，甚至会出现资源不足导致响应卡顿，这也是你应该考虑的一个因素。比如常用的 Filter Query 内部需要为匹配的每个索引文档维护一个 8 字节的 maxDoc，那么如果你的索引文档总数是 16 千万，Filter Query 需要的额外内存大概就是 20M，如果你再多分 4 个 Shard，那么每个 Shard 的 Filter Query 需要的额外内存就减少到 5M 啦！当然前提是你有足够的服务器允许你划分那么多 Shard。

最后你需要考虑的是索引数据未来的增长趋势，如果硬件条件允许的话，你可以按照未来某个阶段的数据量来考虑 Shard 的个数，即留个缓冲的过程。比如你当前搜索程序划分了 4 个 Shard 运行良好，那么此时你应该考虑划分 6 个 Shard 来应对后续的索引量增长。



总之, Shard 划分的总个数需要经过不断测试总结得出, 比如你可以先划分为 2 个 Shard, 测试此时的索引吞吐量和查询耗时, 然后逐步加大 Shard 个数, 直到你开始发现问题, 比如系统抛出 `OutOfMemoryException`, 或者索引创建耗时开始变得缓慢了, 或者查询性能开始下降了, 此时你应该能大致得到一个性能的趋势图, Shard 的个数如何确定我想就不言而喻了。当然你的实际环境可能是几十亿的数据量, 你不可能在测试环境完全真实的模拟产品环境, 所以只能缩小数据量进行测试, 比如你实际数据量为 10 亿, 那么可以测试环境下拿 100 万进行测试, 然后类比到真实环境再粗略估算。这里也没有一个确切的公式供你直接计算, 你能做的就是不断测试。

### 10.5.2 Replicate

如果你的 Solr 集群每秒能够处理 100 个查询, 但是应用程序要求每秒能够处理 150 个请求, 此时你可能希望创建一个完全相同的 Relication 副本, 而不是再添加一个 Shard 分片。从下面的图 10-6 你可以看到所有的索引文档全部发送给 Master 节点进行索引, 原先的每秒 100 个查询请求全部由 Master 节点来处理, 现在我们将 Master 节点上的所有数据全部复制一份放到 Slave 节点上, 两者拥有着相同的数据, 因此 Slave 节点也能平摊一部分查询请求, 此时两台节点一起每秒能够处理的查询请求大约是 200 个 / 秒, 这样就基本满足你的查询性能需求。

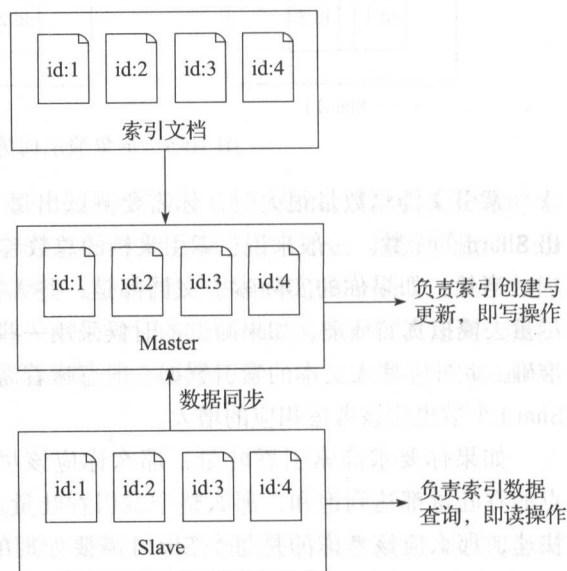


图 10-6 Solr 的 Master-Slave 主从架构

由于 Slave 节点不接收索引创建和更新请求, 所以理论上讲, Slave 节点每秒能够处理的查询请求数应该比 Master 节点要多, 在大部分情况下, Master 一般用于接收索引的创建和更新请求即写操作, 而 Slave 节点一般负责接收索引数据的查询请求即读操作。通过在 Solr 集群中设计这种读写分离架构, 这样使得你的查询请求的性能几乎不受重建索引操作的影响, 尽管索引重建可能会过度负载 Master 节点。当你想要在 Solr 集群中隔离不同索引的查询操作或者你想要提高查询每秒的处理能力时, 可以考虑采用这种 Master-Slave 主从架构。

上面我们通过增加一个 Slave 节点提高了系统的整体查询处理能力, 但是当我们的 Master 或 Slave 其中任意一个节点宕机了会发生什么呢? 当一个节点挂掉了, 首先它会降低查询性能, 然后你剩下的那个节点随时面临着单点故障, 一旦剩下的那个节点也挂了, 那么你整个系统就不可用了, 这并不是我们所期望的, 我们一般期望系统即便在出现某个节点故



障时，在不影响系统性能的同时依然能正常工作。如果你想要这种自动容错机制，那么你需要额外复制一个 Slave 节点，当其中一个节点挂掉，备用的 Slave 节点能立即顶替上。通过利用 Relication 机制，你可以构建多个冗余的副本节点来保证系统不会出现单点故障问题。

现在你已经理解了为什么我们需要副本，那么可以开始了解如何在 Solr 集群中添加一个副本节点。从根本上来讲，副本需要一个节点执行索引创建操作（即 Master），同时需要一个节点从 Master 节点拉取索引数据到当前节点。/replication 这个请求 URL 对应的请求处理器必须提前在 Master 和 Slave 两个节点的 solrconfig.xml 中进行配置，这样 Slave 才能定时的通过 /replication 请求处理器去检查 Master 上的索引数据是否已经更新，如果更新了，则会重新拉取数据更新本地节点的数据从而保持两者数据同步。下面的配置示例演示如何在 solrconfig.xml 中配置一个 replication 副本：

```
// Master 节点的 solrconfig.xml
// (http://masterserver:8080/solr/core1)
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="enable">true</str>
    <str name="replicateAfter">commit</str>
    <str name="replicateAfter">optimize</str>
    <str name="replicateAfter">startup</str>
  </lst>
</requestHandler>

// Slave 节点的 solrconfig.xml
// (http://slaveserver:8080/solr/core1)
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="enable">true</str>
    <str name="masterUrl">
      http://masterserver:8080/solr/core1/replication
    </str>
    <str name="pollInterval">00:00:15</str>
  </lst>
</requestHandler>
```

在 Master 的配置中，replicateAfter 参数用于指示 Slave 节点什么时候进行索引数据复制。若 replicateAfter 参数指定为 commit、optimize、startup 表示当 Master 节点上有索引提交、索引优化、Master 节点启动操作完成之后就触发 Slave 节点去 Master 节点拉取最新的索引数据，从而保证两者之间数据同步。

对于 Slave 端的配置中，则需要指定 masterUrl，因为 Slave 节点需要从 Master 节点拉取数据，所以 Slave 节点需要知道 Master 节点的访问 URL，其中 pollInterval 参数用于指定 Slave 节点每间隔多长时间去访问 Master 节点检测索引数据是否有更新，这个检测频率怎么设置，取决于你耳朵索引数据更新是否频繁，如果索引数据基本一个星期才会更新一次，那么你可以设置为 7 天，如果索引数据基本每秒都会改变，那么你可以设置为 N 秒。最后一个 enable 参数表示启用当前的 Slave 节点。当然此参数如果用于 Master 节点，那么就是表



完成这些复杂的 Solr 集群维护工作。

在 Solr 5.x 中, 默认使用的是 Core 自动发现机制, 主要是通过 Core 主目录下的 `core.properties`, 只要该目录下包含了此文件就会被自动发现并加载。采用这种方式, Core 加载更加自动化, 而且每个 Core 的定义更加的分散, Core 与 Core 之间更加的独立, 这也为 Core 的维护提供了便利。

想要借助 Core 自动发现机制手动添加一个 Core, 你可以阅读第 2 章内容进行回顾学习。当你想要动态的添加一个 Core, 你可以借助 Solr Core Admin API, 发送一个 Create 请求给 Solr Server, 示例如下所示:

```
http://localhost:8080/solr/admin/cores?
action=CREATE&name=coreX&instanceDir=coreX&
config=solrconfig.xml&schema=schema.xml&
dataDir=data&loadOnStartup=true&transient=false
```

对于 Core 的 Create 操作, 只有 `name` 和 `instanceDir` 这两个参数是必须指定的, 如果你要将 `schema.xml` 和 `solrconfig.xml` 指定为其他任意路径或任意文件名称, 你可以指定 `schema` 和 `config` 参数。同理你也可以指定 `data` 参数设置 Core 的数据目录。`loadOnStartup = true` 表示是否在 Solr Server 启动时就加载 Core, 否则就延迟加载 Core。`transient = true` 表示当系统内存紧张需要释放资源回收内存时或者 Solr Server 重启之后, 会将 `transient = true` 的 Core 给卸载掉, 即你的 Core 没有被持久化, 只是存在于内存中。



**注意** 任何你通过 Solr Web 后台的 Core Admin 添加的 Core 都是瞬态的即 `transient = true`, 当你 Solr Server 重启之后, Core 就被卸载了。

如果修改了你的 Core 的配置文件 (比如 `solrconfig.xml`、`schema.xml`, 或者 `conf` 目录下的其他文件), 这些配置更新并不会自动生效, 你需要重新加载 Core 才能使配置更新生效, 此时你需要对你的 Core 执行 Reload 操作, 具体请查阅第 2 章内容。但是需要注意的是, Core 的 `dataDir` 参数以及 `solrconfig.xml` 中 `IndexWriter` 类相关的配置参数, 你即便重新加载了 Core 也不会生效, 此时你只能重新启动 Solr Server。Solr Core 重启之后, 关于该 Core 的缓存和 Request Handler 收集的统计信息将会丢失。

有时候, 你可能想要修改某个 Core 的名称, 那么你可以对 Core 执行 Rename 操作。通过指定 Core 和 `other` 参数即可 Core 的重命名。但是需要注意的是, 在 Core 重命名过程中, 该 Core 无法被访问, 这也是它跟 SWAP 操作的最大区别。另外还需要清楚的一点就是, Core 重命名修改的其实是 `core.properties` 文件中的 `name` 属性值, 并不会真正去修改你硬盘上 Core 文件夹名称。

此外, 有时候你可能期望卸载某个 Core, 你需要对 Core 执行 Unload (具体操作请看第 2 章) 操作。但是需要注意的是, 卸载一个 Core 并不会真正地在硬盘上删除该 Core 的索引

数据和 Core 的任何相关配置文件和目录, Core Unload 操作仅仅只是将 Core 从内存中剔除掉, 所谓 Core Load 加载操作其实就是加载硬盘上 Core 的配置文件以及索引文件并放入内存中, 后续需要访问 Core 的一些详细信息其实都是从内存中获取。当然你也可以直接删除硬盘上 Core 的整个根目录来进行卸载, 但一般不建议这么做。

当你的 Core 下的索引数据越来越大时, 你可能会期望将其分割成多个 Core。但是需要注意的是, 你的 targetCore 需要已经存在, 如果 targetCore 在 Solr 中不存在, 那么你可以提前通过 Solr Core 的 Create 操作进行创建。但是如果你暂时不想创建 Core, 那么此时你还可以通过指定 path 参数先将索引数据分割到指定的目录下, 示例如下:

```
http://localhost:8080/solr/admin/cores?action=SPLIT&core=oldCore&
path=/path/to/newCore1/data/&path=path/to/newCore2/data/&
path=path/to/newCore3/data/
```

path 参数表示新 Core 的索引数据目录, 但是此时并不要求新 Core(即这里的 newCore1/newCore2/newCore3) 必须存在, 只需要 path 参数指定的目录在硬盘上存在即可, 这样 oldCore 的索引数据就会被分割到这 3 个目录下, Split 操作完成之后, 你可以再执行 Create 操作创建这 3 个 Core。同理, Solr Core 的 MERGEINDEXES 操作即可以直接指定多个 srcCore 参数进行多个 Core 的合并, 也可以通过指定 indexDir 参数将多个索引目录下的索引数据合并到新 Core 中。

## 10.7 Solr 集群管理

当你的 Solr 实例越来越多时, 管理 Solr 集群也变得越来越复杂。当其中某个节点宕机, 可能会直接影响其他节点的正常运行。你需要有一种方式能够实时监测到集群中每个节点的健康状态。当你不断地向 Solr 集群中添加新的节点, 如何高效的管理每个节点上的配置文件?

### 10.7.1 Solr Ping 健康检测

如果你工作的公司有幸有硬件负载均衡器, 那么你可能已经知道如何为你的每个节点设置虚拟 IP 地址或者你们公司有专门的人员负责这块。如果你使用的是云服务器, 那么一般你可以使用云服务器提供的弹性负载均衡器来建立负载均衡。如果你工作的公司是个小作坊, 那么你可以安装一个 HAProxy 来作为负载均衡器。无论你选择使用哪种解决方案, 你都希望负载均衡器能够监测 Solr 集群中每个节点的健康状态信息, 从而帮助你判断 Solr 集群中每个节点的健康状态。为了实现节点的健康检查, Solr 提供了 Ping Request Handler, 它需要在 solrconfig.xml 中稍作配置进行启用, 配置示例如下所示:

```
<requestHandler name="/admin/ping" class="solr.PingRequestHandler">
  <lst name="invariants">
    <str name="q">title:xxxx AND content:xxxxxxx</str>
    <str name="shards">
```

```
localhost:8080/solr/core1,remotehost:8080/solr/core2
</str>
</lst>
<str name="healthcheckFile">server-enabled</str>
</requestHandler>
```

如果你启用了 Ping Request Handler, 那么负载均衡器可以向该节点发送 ping 请求:

```
http://localhost:8080/solr/core1/admin/ping
```

你还可以指定 healthcheckFile 参数, 此参数表示如果 healthcheckFile 参数指定的健康检测文件 server-enabled 在当前 Core 的 conf/ 目录下不存在的话, 那么 ping 请求会直接返回错误状态码, 此时负载均衡器就认为此节点出问题了, 将其排除在 Solr 集群中。通过指定 healthcheckFile 参数, 使得你可以不需要手动去关闭集群中某个节点, 你只需要删除 healthcheck-File 参数指定的文件即可。你可以通过如下请求来创建或删除 health check 文件以及启用禁用 ping:

```
// 启用 ping, 若配置了 healthcheckFile 参数, 且 health check 文件不存在那么会自动新建
http://localhost:8080/solr/core1/admin/ping?action=enable
// 禁用 ping
http://localhost:8080/solr/core1/ping?action=disable
// 检查 health check 文件是否存在
http://localhost:8080/solr/core1/admin/ping?action=status
// 执行 PingRequestHandler 中 q 参数执行的查询并返回执行是否成功,
// 若配置了 healthcheckFile 参数, 还会检查 health check 文件是否存在
http://localhost:8080/solr/core1/admin/ping
```

## 10.7.2 Solr 配置文件管理

Solr 具有高度可定制化特性。你可以单独为你的每个 Core 分别指定 schema.xml 和 solr-config.xml, 然后定义硬编码在 schema.xml 中, 这对于小规模 Solr 部署来说还能很好的工作。一旦你的 Solr 节点达到了几十几百个, 每个配置文件可能都不太一样, 管理这些配置文件变得更加艰难。每天你升级或者重新部署 Solr, 你都需要手动的去更新升级每个节点上的配置文件 (solrconfig.xml、schema.xml 等)。可以减少你的工作量的一种策略就是尽可能地采用统一的配置文件。比如企业构建基于云计算的 Solr 搜索服务, 经常需要定义各种域类型, 允许用户在任何时刻在不需要修改 schema.xml 的前提下自定义新的域。在这种情况下, 每个 Solr Server 需要拥有相同的 schema.xml 文件, 这意味着重新部署集群中任何一个节点等同于重新部署整个 Solr 集群。采用统一的配置文件可能会带来一些问题, 最大的问题就是我们必须提前在公共的 schema.xml 中定义所有动态域以及动态域用到的域类型。这就需要你进行大量的预先思考。其实, 这种方式本质还是充分利用 Solr 的动态域特性, 你可以提前定义很多动态域以满足未来各种动态新增域, 但是你不想要频繁的修改 schema.xml 时, 那么就必须提前定义很多的动态域, 以保证后续新增的域都能直接应用动态域, 这要求你必须提前考虑周全, 如果你新增的域无法应用动态域, 那么可能会抛异常。

尽管采用统一的配置文件使得配置文件的管理变得简单了，但是采用动态域性能会稍微差些。此外采用统一的配置没法满足多样化的需求，比如数字域的 `precisionStep` 属性没法统一进行设置。采用动态域的方式也不利于阅读，因为域名称都是采用通配符的方式进行模糊匹配的，开发的时候给人感觉不太直观。当然，你可以针对 `solrconfig.xml` 采用统一的配置文件，这看起来更有挑战性。因为不同的节点需要的配置以及需要加载的其他文件都不同，但是你仍然可以通过 `core.properties` 来实现。

比如你配置 `ReplicationHandler` 需要指定相关参数，但是每个节点的值可能会各不相同，此时你可以自定义的变量来代替，这样每个节点的配置都是一致，至于变量的具体值你可以在各个 Core 的 `core.properties` 文件中进行指定，具体示例如下所示：

```
// 统一的 solrconfig.xml
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="enable">${master.replication.enabled:false}</str>
    <str name="replicateAfter">commit</str>
    <str name="replicateAfter">optimize</str>
    <str name="replicateAfter">startup</str>
  </lst>
  <lst name="slave">
    <str name="enable">${slave.replication.enabled:false}</str>
    <str name="masterUrl">
      http://${masterserver:}/solr/${solr.core.name}/replication
    </str>
    <str name="pollInterval">00:00:15</str>
  </lst>
</requestHandler>
// Slave 节点的 core.properties 文件
Slave server' s solrcore.propertiesfile
...// 其他属性省略
slave.replication.enabled="true"
masterserver="masterserver:31000"
// Master 节点的 core.properties 文件
Master server' s solrcore.propertiesfile
...// 其他属性省略
master.replication.enabled="true"
```

你会注意到，在自定义变量的时候，你可以通过冒号后面跟随指定一个默认值，表示当你 `core.properties` 中没有显式的指定变量值时就采用冒号后面的默认值。这样，你 Solr 集群中的任意一个节点都可以共享这一份 `solrconfig.xml` 配置文件即可了，后续你只需要维护这一份 `solrconfig.xml` 配置文件，极大减少了你的工作量。

## 10.8 如何与 Solr 交互

尽管我们之前在讲解 Facet 和 Group 查询等内容时介绍了如何与 Solr 进行交互实现查



询。但是 Solr 还提供了其他与 Solr 交互的方式。本章节将重点讲解如何实现你的项目与 Solr 进行交互的 N 种方式。

### 10.8.1 使用 REST API 与 Solr 交互

REST API 是最常见的与 Solr 进行交互的方式。因为 REST API 仅仅只是依赖于 Http 协议，只要你的项目能够通过网络连接到 Solr Server，那么你就可以通过 REST API 实现与 Solr 交互。

尽管 Solr REST API 返回的响应数据支持多种格式，但是对于 Solr 初学者而言，使用自己喜爱的编程语言来向 Solr 发送一个请求可能会是一件很痛苦的事情。不过前辈们早已封装了很多类库帮助你以面向对象的方式使用 REST API 与 Solr 交互。这部分内容第 13 章会做详细介绍。

尽管 Solr REST API 已经暴露了 Solr 的所有可用功能，也有很多客户端类库将 REST API 采用面向对象的方式进行了封装。但是并不是所有的类库都支持 Solr 最新的发布版本，但是 Solr 在 REST API 的向后兼容性方面考虑的十分周全，这意味着即便使用的客户端类库版本已经过时了，但是你仍然能够通过它访问 Solr 的大部分功能。目前比较流行的 Solr 客户端类库包括 SolrJ (Java)、RSolr (Ruby)、Solarium (PHP)、ScalaLikeSolr (Scala)、SolPython (Python)、SolJSON (JavaScript)、SolrNet (.NET)、SolPerl (Perl)。当然 Solr 的客户端类库支持的编程语言远不止这里我列举的这些，详情请访问如下链接进行了解：

<http://wiki.apache.org/solr/IntegratingSolr>

因为 Solr 是采用 Java 语言编写的，SolrJ 允许你直接使用二进制数据格式与 Solr 进行数据通信，这也是 SolrJ 的默认格式（即 javabin，因为二进制形式数据传输性能比较高）。SolrJ 具体如何使用会单独开辟一章进行详细讲解。

### 10.8.2 使用 SolrJ 与 Solr 进行交互

使用 SolrJ 可以很方便的以面向对象的方式与 Solr Server 进行数据通信，当你下载了 Solr 安装包，SolrJ 的 jar 包已经包含在内。之所以使用 SolrJ，是因为它支持 Java 二进制形式的请求 / 响应格式，它还减少了其他数据格式（如 JSON/XML）的序列化和反序列化的执行开销。使用 SolrJ，你可以通过 Http 连接与 Solr Server 进行交互，你还可以直接将 Solr 嵌入到你当前项目，然后你不需要在外部暴露 Solr 的 Http 接口，直接在你的项目内部通过 SolrJ API 与 Solr 进行交互。当然如果你项目使用的是 PHP 语言又或者你更擅长 PHP 这门编程语言，那么可以选择其他 Solr 客户端类库。

SolrJ 的 jar 包你可以在 solr-5.3.1\dist 目录下找到，当然你也可以自己在网络上下载或者使用 Maven 方式获取到。SolrJ 自身包含了对其他类库的依赖，SolrJ 依赖的相关 jar 包在 solr-5.3.1\dist\solrj-lib 目录下。如果项目中使用了 Maven，那么导入 SolrJ 依赖就显得非常

简单：

```
<dependency>
<groupId>org.apache.solr</groupId>
<artifactId>solr-solrj</artifactId>
<version>5.3.1</version>
</dependency>
```

当你将 SolrJ 依赖的 Jar 包导入到了项目中之后，你需要创建 SolrClient（旧版本 SolrJ 里使用的是 SolrServer）对象与 Solr Server 进行交互。你可以创建 HttpSolrClient 或者 EmbeddedSolrServer 或者 ConcurrentUpdateSolrClient 对象，通过该对象你可以连接到 Solr Server。SolrClient 抽象类内置有很多实现，如图 10-8 所示。



表 10-1 列举了 SolrClient 的各种实现类及其作用。

图 10-8 SolrClient 的类继承关系

表 10-1 SolrClient 实现类

SolrClient 实现类	描 述
HttpSolrServer	用于客户端与 Solr Server 进行 Http 通信的类，不过目前此类已提示被废弃，不建议使用了，推荐使用 HttpSolrClient 代替 HttpSolrServer
HttpSolrClient	跟 HttpSolrServer 类似，注意：以 SolrServer 结尾的类都已经不推荐使用了。此类基本上是线程安全的（因为内部有个 setParser 方法不是线程安全的），但不适用于 SolrCloud 模式下
ConcurrentUpdateSolrClient	用于发送索引 update 和 delete 请求，它会采用队列的方式批量提交更新的索引文档以提高索引吞吐量。此类是线程安全的。它其实就是 HttpSolrClient + 线程池
LBHttpSolrClient	在 Solr 集群的多个节点之间启用请求的负载均衡。在 Master-Slave 主从架构下，请不要使用此类用于索引创建、更新和删除操作，因为索引数据只能发送给 Master。在 SolrCloud 的 leader/replicate 架构下，通常建议你使用 CloudSolrClient，但是你也可以使用此类来执行索引 update 操作，此类会自动将请求转向正确的 Leader 节点。此类提供了自动故障转移功能，即当一个节点挂掉了，它能将请求转移到集群中的其他节点。当节点又重新恢复正常了，它又能自动检测到并将其加入到 Solr 集群中。负载均衡采用的是简单的集群节点列表遍历。当请求某个节点抛出 IO 异常导致请求失败了，可能是由于 Http 连接超时或者网络读操作超时，那么该节点就会从活跃节点列表中被剔除出来，同时会被加入到一个已挂掉的节点列表中，然后请求会被转发到下一个活跃节点，此过程会一直持续直到找到一个活跃节点为止。LBHttpSolrClient 内部会起一个检测线程按照固定的检测周期去检测已经挂掉的节点是否又“活过来了”。你可以通过 setAliveCheckInterval 方法来设置检测间隔周期，默认间隔是 60 秒。此类的替代品有专门的硬件负载均衡器或者 Apache Http Server 的 mod_proxy_balancer（具体请访问 <a href="http://en.wikipedia.org/wiki/Load_balancing_(computing)">http://en.wikipedia.org/wiki/Load_balancing_(computing)</a> 进行详细了解）。此类是线程安全的

(续)

SolrClient 实现类	描 述
CloudSolrClient	用于跟 SolrCloud 进行通信的客户端类，内部会先通过此类请求 Zookeeper，通过 Zookeeper 自动发现节点，然后通过 LBHttpSolrClient 去发送请求。此类是线程安全的
EmbeddedSolrServer	在不启用 Http 访问的情况下，在当前系统中嵌入 Solr，然后通过此类去访问 Solr 服务。这种嵌入式访问模式一般用于开发测试阶段。此类不支持分布式查询，不支持同一台服务器上跨多个 Core 查询

你可以借助 SolrJ 来提交索引文档以及发送查询请求到 Solr Server，SolrJ 可以直接发送大部分的 Solr 请求。我们之前的第 7 章在导入测试数据时，就一直演示了如何使用 SolrJ 来提交索引文档到 Solr Server，而后 Solr Server 会在服务器端建立索引。下面是一个 SolrJ 的简单使用示例，如下所示：

```

SolrServer server = new HttpSolrServer("http://localhost:8080/solr/core1");
server.deleteByQuery("*:~");
SolrInputDocument doc1 = new SolrInputDocument();
doc1.addField("id", "1", 1.0f);
doc1.addField("cat", "health", 1.0f);
doc1.addField("price", 100);
SolrInputDocument doc2 = new SolrInputDocument();
doc2.addField("id", "2", 1.0f);
doc2.addField("cat", "entertainment", 1.0f);
doc2.addField("price", 150);
SolrInputDocument doc3 = new SolrInputDocument();
doc3.addField("id", "2", 1.0f);
doc3.addField("cat", "entertainment", 1.0f);
doc3.addField("price", 99);
Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
docs.add(doc1);
docs.add(doc2);
docs.add(doc3);
server.add(docs);
server.commit();
SolrQuery query = new SolrQuery();
query.setQuery("*:~");
query.addSortField("price", SolrQuery.ORDER.desc);
QueryResponse rsp = server.query(query);
SolrDocumentList docs = rsp.getResults();
SolrQuery solrQuery = new SolrQuery()
    .setQuery("*:~").setFacet(true).setFacetMinCount(1).setFacetLimit(10)
    .addFacetField("cat");
QueryResponse rsp = server.query(solrQuery);
server.deleteByQuery("*:~");

```

你除了可以使用 SolrJ 通过 HTTP 与 Solr Server 进行交互，还可以直接将 Solr 嵌入到另一个 Java Project 中，然后借助 EmbeddedSolrServer 在内部直接连接 CoreContainer 实现与

Solr 的交互。下面是 `EmbeddedSolrServer` 的简单使用示例，如下所示：

```
String solr_home = "/path/to/solr/home";
CoreContainer container = new CoreContainer(solr_home);
container.load();
EmbeddedSolrServer server = new EmbeddedSolrServer(
    container, "core1" );
SolrQuery query = new SolrQuery();
query.setQuery( "*" );
QueryResponse rsp = server.query( query );
SolrDocumentList docs = rsp.getResults();
```

`EmbeddedSolrServer` 的使用与其他 `SolrClient` 实现类的最大区别就是在构建 `EmbeddedSolrServer` 时你需要传入 `SOLR_HOME` 的路径，以及你要访问的 Core 的名称。`EmbeddedSolrServer` 相对于其他实现有个很大的缺点就是它不支持分布式查询。此外你还不能在同一台服务器上跨多个 Core 进行查询。

`SolrJ` 是向后兼容 `Solr` 的，只要 `SolrJ` 的主版本与 `Solr` 版本保持一致即可。但是你需要注意的是 `SolrJ` 很早之前与 `Solr` 通信采用的是 XML 格式来请求和响应数据，在最新版本中，`SolrJ` 将默认的数据格式从 XML 转变成 `JavaBin`（即二进制格式），作出这种改变主要为了降低网络间数据传递过程中的序列化和反序列化的执行开销。

## 10.9 监控你的 Solr

当你将 `Solr` 部署到真实的产品环境中，了解 `Solr` 集群的运行性能非常重要。如果在系统负载还不是很高的时候 `Solr` 查询速度过慢，那么你可能需要进一步对你的索引数据进行分片，降低查询的复杂度或者增加服务器的资源（比如加内存）。如果你的查询流量越来越大，那你可能需要为节点再添加几个索引副本以降低查询的负载。如果你的查询未能达到最佳的缓存性能或者查询出现了异常，那么此时需要对代码进行调试。本章节将会深入介绍 `Solr` 内部提供的性能统计和 `Solr` 的日志文件。

### 10.9.1 Solr 的性能统计

在 `Solr` 的后台管理界面，当你选择了一个 Core 之后，在最左侧你会看到一个“Plugins / Stats”菜单，单击后你会在右侧看到 `Solr` 中各种组件的运行性能状态统计数据，其中包括 Core、Cache（缓存）、`QueryHandler`（查询处理器）、`QueryParser`（查询语法解析器）、`UpdateHandler`（索引更新处理器）等组件的性能统计。

单击 Core 统计部分，你会发现右侧展示了 3 部分统计信息：注册的 `IndexSearcher` 实例相关信息以及当前 Core 相关信息。其中有两个 `IndexSearcher` 实例是因为你硬提交之后就会重新 `New` 一个新的 `IndexSearcher` 实例。其中列举的统计信息基本上根据名称就应该知道表达的含义，我只想提一点，那就是 `warmupTime`，它表示当前 `IndexSearcher` 实例 `AutoWarm`

(自动预热)花费的时间,从这个数据你大致能够判断当前 IndexSearcher 运行是否正常,同时你还要注意 caching 是否等于 true 即当前 IndexSearcher 实例是否开启了缓存。

### 10.9.2 Solr 的缓存性能

Solr 的缓存决定了 Solr 的性能。单击中间的“CACHE”,你会在右侧看到当前 Core 应用了哪些缓存,以及每种类型的缓存的使用情况。其中 lookups 表示你查询了几次缓存; hits 表示缓存命中了几次; hitratio 表示缓存命中率; inserts 表示你往缓存中插入了几个缓存对象; evictions 表示有几个缓存对象被从缓存中“踢”出来了; size 表示缓存空间大小; warmupTime 表示缓存预热花费的时间。其中 hitratio 和 warmupTime 是两个非常重要的性能指标, hitratio 值越大说明缓存命中率越高,也就意味着你的大部分查询都是直接从内存中返回的,说明你的查询性能很好,反之即说明你的查询性能很差。warmupTime 决定了你的缓存预热时间,缓存预热时间越长那么你的查询实时性就越差。此外你还需要根据 hitratio 和 warmupTime 的统计值来判断是需要增大缓存空间呢还是减小缓存空间。此时你需要考虑的是那些使用比较频繁的查询返回的结果集需要占用的缓存空间有多大。这你就需要通过日志记录来统计出那些经常被用户使用到的查询,比如统计出 Top100 或者 Top 500 的查询,然后估算出这些查询返回的一页数据所需要占用的内存大小,然后再稍微留一些剩余空间,这样你就能大致估算出缓存空间大小,当然你不可能一次性就估算准确,需要慢慢调整。切记,把那些用户几乎很少用到的查询匹配到的结果集放入缓存中是毫无意义的。

### 10.9.3 Solr JMX

在 Java 程序的运行过程中,对 JVM 和系统的监测一直是 Java 开发人员在开发过程所需要的。一直以来,Java 开发人员必须通过一些底层的 JVM API,比如 JVMPi 和和系统的一系列情况,这种方式一直以来被人所诟病,因为这需要大量的 C 程序和 JNI 调用,开发效率十分低下。于是出现了各种不同的专门做资源管理的程序包。为了解决这个问题, Sun 公司也在其 Java SE 5 版本中,正式提出了 Java 管理扩展 (Java Management Extensions, 简称 JMX) 用来管理检测 Java 程序 (同时 JMX 也在 J2EE 1.4 中被发布)。

JMX 既是一套标准或者规范,又是一个框架,它定义了管理系统与操作系统资源之间交互的接口。JMX 的提出,将在 JDK 中开发自检测程序变成了可能,它也提供了大量轻量级的检测 JVM 和运行中对象/线程的方式,从而提高了 Java 语言自身的管理监测能力。关于 JMX 的详细信息请读者自行通过 Google 搜索“JMX”搜集学习资料然后自己学习了解,因为本书也不是重点讲解 Java 基础的。

Solr 中天然支持 JMX。自从 Solr 1.3 版本开始, Solr 就已经支持以动态 MBean 的形式暴露运行时统计数据,进而你可以使用 JMX 客户端 (比如 jconsole) 来建立自己的 Solr 监视器。自 Solr 3.1 版本开始,这些统计信息你还可以通过 Http 请求 SolrInfoMbeanHandler 来获取, SolrInfoMbeanHandler 默认被映射的请求 URL 为 <http://localhost:8080/solr/core1/>



admin/mbeans/, 其实你在 Solr Web 后台的“Plugins / Stats”中看到的统计信息就是通过 SolrInfoMbeanHandler 获取的。

要在 Solr 中启用 JMX, 你只需要在 solrconfig.xml 文件中配置 `<jmx/>`。当且仅当 MBeanServer 存在了, JMX 才会被启用。如果你配置了 `<jmx/>`, 那么 Solr 会列举出所有的可用 MBeanServer, 使用第一个 MBeanServer 将其注册为 MBean。你也可以为 MBeanServer 指定一个唯一的 agentId, 比如:

```
<jmx agentId="myMBeanServer" />
```

你也可以为你的 MBeanServer 暴露一个远程访问的 URL, 这样你能远程访问 MBeanServer 进行监控了, 比如:

```
<jmx serviceUrl="service:jmx:rmi:///jndi/rmi://localhost:9999/solrjmx" />
```

一切配置就绪之后, 你就可以直接双击 JDK 的 bin 目录的 jconsole.exe 应用程序, 然后你会看到连接界面, 暂时先选择本地连接, 然后直接如图 10-9 双击 Tomcat 进程 (包含 org.apache.catalina 的那个是 Tomcat 进程) 那一项就能开始连接 MBeanServer, 如果提示“安全连接失败”, 那么请选择“不安全”连接。连接成功之后, 你会看到如图 10-9 所示的界面。

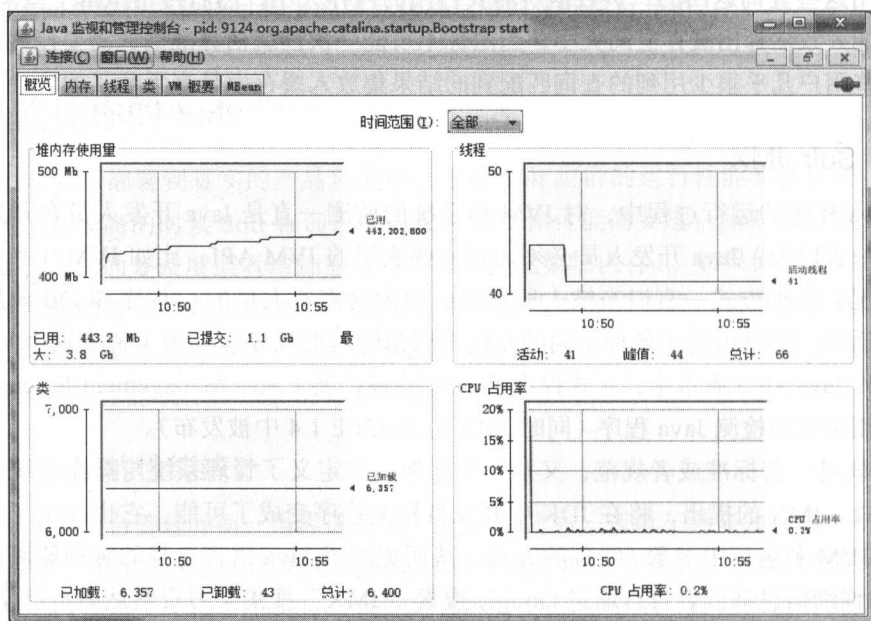


图 10-9 Java 监视管理控制台界面

概览中采用图表的形式形象展示了堆内存使用情况、CPU 占用率、类加载情况、活跃线程情况等信息, 切换到“内存”选项卡, 这里同样以图表的形式展示了 JVM 内存各个部分的使用情况, 显示的信息更为详细。最后一项“MBean”里是 Tomcat 和 Solr 注册的



MBean。你在 Solr Web 后台里能看到的统计信息，在“MBean”里都能看到，这里是采用树形菜单的形式展现，层级太多感觉查看起来不太方便。

如果你想要连接远程的 Tomcat 暴露的 MBeanServer，那么你首先需要在 Tomcat 的 catalina.bat（Linux 环境下就是 catalina.sh）文件中启用 JMX MBeanServer 远程访问。打开 tomcat bin 目录下的 catalina.bat 文件，然后如下进行配置：

```
rem ----- Execute The Requested Command -----
set JAVA_OPTS=%JAVA_OPTS% -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.ssl=true
-Dcom.sun.management.jmxremote.authenticate=true
-Djava.rmi.server.hostname=localhost
echo Using CATALINA_BASE: "%CATALINA_BASE%"
```

如果是 Linux 环境下，那么配置如下：

```
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.ssl=true
-Dcom.sun.management.jmxremote.authenticate=true
-Djava.rmi.server.hostname=localhost"
# ----- Execute The Requested Command -----
```

- ❑ com.sun.management.jmxremote 表示开启 JMX MBeanServer 远程访问；
- ❑ com.sun.management.jmxremote.port 表示设置 JMX connector 监听的端口号；
- ❑ com.sun.management.jmxremote.ssl 表示是否启用 Http SSL 安全连接；
- ❑ com.sun.management.jmxremote.authenticate 表示是否启用账号密码安全认证，账号密码默认从 JAVA\_HOME/jre/lib/management 下的 jmxremote.access 和 jmxremote.password 两个文件中获取。jmxremote.access 文件用于定义账号和该账号所拥有的权限。jmxremote.password 文件用于定义账号对应的登录密码。当然你也可以通过 com.sun.management.jmxremote.access.file 和 com.sun.management.jmxremote.password.file 这两个参数来指定 jmxremote.access 和 jmxremote.password 这两个文件的加载即路径来覆盖默认 JDK 提供的两个文件；
- ❑ com.sun.management.jmxremote.authenticate 表示是否需要开启登录验证；
- ❑ java.rmi.server.hostname 用于指定 JMX MBeanServer 的域名或者 IP，当你的 JMX MBeanServer 主机的网络采用的是 NAT 模式，那么此时 java.rmi.server.hostname 参数会很有用。

jmxremote.access 文件配置示例如下所示：


```
admin  readwrite \
      create javax.management.monitor.*,javax.management.timer.* \
      unregister
```

这里定义了一个 admin 账号，后面是该账号拥有的权限，readwrite 表示拥有读写权限，同理还有 readonly 表示只有读权限。unregister 表示赋予账号拥有删除 MBean 的权限，“create”关键字表示赋予账号拥有创建指定包路径下的 MBean 权限。

jmxremote.password 文件配置示例如下所示：

```
admin 123
```

前面是账号，后面表示密码，中间使用两个空格字符进行分割。如果你没有配置 com.sun.management.jmxremote.authenticate 或者你将 com.sun.management.jmxremote.authenticate 设置为 false，那你不需要配置 jmxremote.access 和 jmxremote.password 这两个文件。

 注意

linux 环境下你需要赋予你 Tomcat 启动用户拥有访问 jmxremote.access 和 jmxremote.password 两个文件的权限：  
chmod 600 jmxremote.access;  
chmod 600 jmxremote.password.

配置完成之后，你需要重启 Tomcat 使其立即生效，然后在 jconsole 的新建连接界面就需要选择“远程连接”，然后采用 service:jmx:rmi 格式的 URL（当然你也可以直接 ip: port 方式）进行远程连接，如图 10-10 所示。

你除了可以通过配置 JMX 然后借助 JDK 提供的 jconsole 工具来访问 JMX MBeanServer，从而能实时监控 Solr 的性能之外，你还可以通过 Solr 内置的 SolrInfoMbeanHandler 通过 HTTP 请求直接访问 Solr 各组件的性能统计数据。Solr Web 后台的“Plugins / Stats”菜单功能使得 Solr 用户能够非常便捷地以页面可视化的方式查阅 Solr 各组件的性能统计数据，SolrInfoMbeanHandler 使得用户能够以 REST API 编程的方式通过构造 HTTP 请求来获取 Solr 各组件的性能统计数据。SolrInfoMbeanHandler 可以接收以下请求参数，如表 10-2 所示。

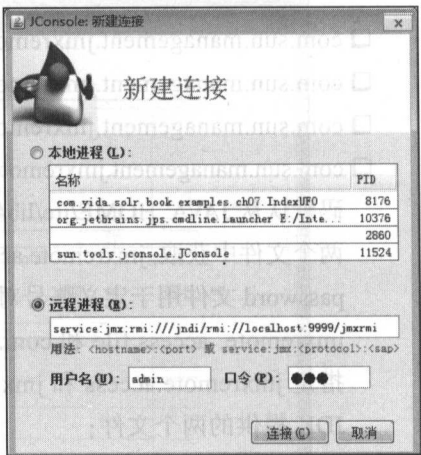


图 10-10 JMX 远程连接

表 10-2 SolrInfoMbeanHandler 参数表

参数	默认值	描 述
key	所有 key	指定在返回结果集中包含哪个 MBean 的信息，这里 key 表示该 MBean 的唯一标识名，若 key 参数未指定，则会返回所有 MBean 的信息
cat	所有分类	表示在返回的结果集中包含哪些分类的信息，每个 MBean 都属于一个分类，一个分类下可能有多个 MBean。可选的分类有 CACHE、CORE、QUERYHANDLER、QUERYPARSER、UPDATEHANDLER，若 cat 参数未指定，则会返回所有分类的信息

(续)

参数	默认值	描 述
stats	false	表示是否统计数据需要随结果集一起返回，此参数可以针对每个 key 进行单独设置，比如： f.fieldCache.stats = false
wt	xml	表示返回的结果集数据输出格式，默认为 XML。

以下是 MBean Request Handler 的几个简单请求示例：

```
// 返回关于 CACHE 分类下的 MBean 信息
http://localhost:8080/solr/core2/admin/mbeans?cat=CACHE
// 返回关于 CACHE 分类下的 MBean 信息以及统计信息，并以 JSON 格式返回
http://localhost:8080/solr/core2/admin/mbeans?stats=true&cat=CACHE&indent=true
&wt=json
// 返回所有 MBean 信息以及统计信息，但 key=fieldCache 的 MBean 除外
http://localhost:8080/solr/core2/admin/mbeans?stats=true&f.fieldCache.
stats=false
// 只返回 key=fieldCache 的 MBean 信息以及统计信息
http://localhost:8080/solr/core2/admin/mbeans?key=fieldCache&stats=true
```

Solr JMX 仅仅只是比较简单的性能监控，更专业的性能监控工具请选择 SPM，SPM 官网地址如下所示：

<http://sematext.com/spm/>

据官网介绍，它支持对如图 10-11 所示的产品进行性能监控。

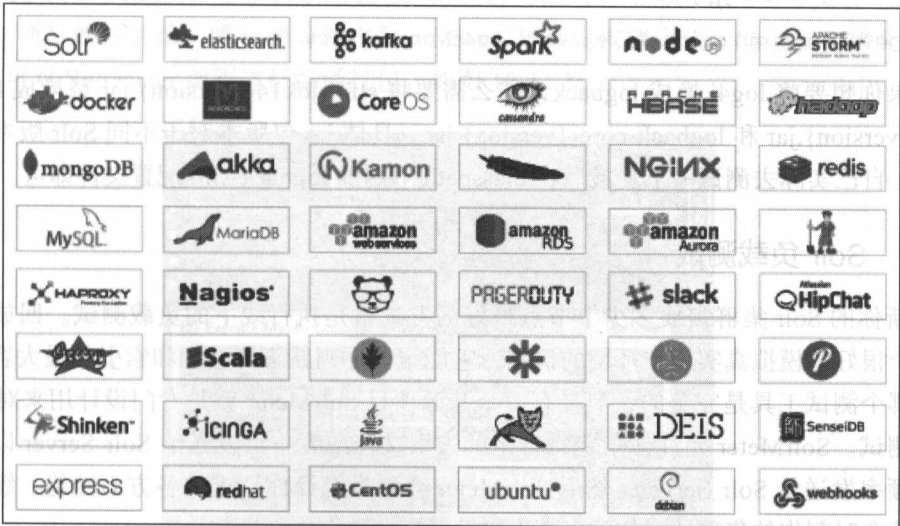


图 10-11 SPM 支持的开源项目

然而遗憾的是，它不是开源的，但是它确实是一个不错的选择。在性能监控的开源项

目中你可以试试 Zabbix。安装 Zabbix Server 之后，你可以借助 Zapcat 工具将自己的 Java 程序当作 Zabbix agent，然后你可以直接在 Java 程序中获取 Zabbix Server 提供的性能监控信息，而不需要使用脚本或者命令行方式来获取，Zapcat 相当于 Zabbix Server 与 JMX 这两种技术之间的桥梁。关于 Zabbix 和 Zapcat 如何安装与部署，请访问各自官方网站查阅相关资料自行学习，这里就不展开了。

### 10.9.4 Solr 日志

与大多数应用程序一样，Solr 的日志为你提供了关于你的集群任何时刻的运行状态的丰富信息。日志信息的打印证明了该查询被执行了，日志信息还会显示每个请求的请求参数，以及每个查询请求的执行时间，以及各种异常信息、请求阶段的一些重要操作记录信息。通过日志信息你还可以了解到 Solr Server 接收到索引文档的时刻，索引文档被提交的时刻、段文件被合并的时刻、索引复制开始和结束时间等。

在 Solr 中启用日志功能，首先需要导入日志依赖的 jar 包，你可以从 solr-5.3.1\server\lib\ext 目录下获取到，然后将其复制到 Solr.war 包解压后的 apache-tomcat-7.0.55\webapps\solr\WEB-INF\lib 目录下，或者直接复制到 apache-tomcat-7.0.55\lib 目录下也是可以的，不过复制到 Tomcat 的 lib 目录下是全局生效的，为了避免对 Tomcat 下其他项目的影响尽量放到 solr 自身的目录下比较好。最后你需要将 log4j.properties 文件放置在 classpath 路径下，如果 Solr 是部署在 Tomcat 下的，那就是 apache-tomcat-7.0.55\webapps\solr\WEB-INF\classes 目录下。或者你可以通过系统参数方式来指定 log4j.properties 文件的加载路径，如下所示：

```
-Dlog4j.configuration=file:/etc/log4j.properties
```

如果你想要将 log4j 换成 logback，那么需要将 slf4j-jdk14-{version}.jar 替换成 logback-classic-{version}.jar 和 logback-core-{version}.jar。具体 jar 包版本对于不同 Solr 版本可能会不一样，自己实际去测试一下。最后在 classpath 下添加 logback.xml 配置文件即可。

### 10.9.5 Solr 负载测试

判断你的 Solr 集群需要多少个节点最好的方式就是执行线下的负载测试。回放历史日志是一个很好的模拟真实用户环境的机制，它能帮助你判断集群查询和索引的最大容量。尽管没有哪个测试工具是完美的，但是有一个开源项目 SolrMeter 就是专门设计用来对 Solr 进行负载测试。SolrMeter 允许你指定将特定的索引文档或者查询发送给 Solr Server 以及以什么样的频率发送给 Solr Server，然后 SolrMeter 会测量统计你请求的各方面性能、缓存利用率以及某个时刻你的集群的详细信息（比如索引提交和优化信息）。

想要使用 SolrMeter，首先你需要从 Github 上下载 SolrMeter 的源码，SolrMeter 项目的 Github 地址如下所示：

`https://github.com/tflobbe/solrmeter`

如果本地已经安装了 Git 客户端环境，那么你可以直接通过 git 命令将 SolrMeter 源码克隆到本地，执行命令如下所示：

```
git clone https://github.com/tflobbe/solrmeter.git
```

或者采用 SSH 连接进行克隆：

```
git clone git@github.com:tflobbe/solrmeter.git
```

或者你直接访问 SolrMeter 的 Github 地址通过网页浏览器进行下载，如图 10-12 所示。

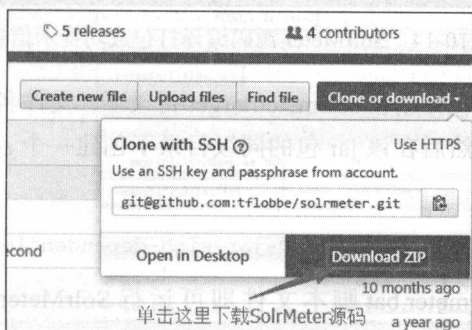


图 10-12 下载 Github 上 SolrMeter 源码

获取到 SolrMeter 的源码之后，请解压到任意盘符路径下，然后打开你的 IDEA（或者 Eclipse，我习惯用 IDEA，所以这里以 IDEA 为例进行说明），依次 File --> Open，然后选择 SolrMeter 项目的根目录，漫长的等待之后，Maven 初始化构建完成。Maven 初始化构建完成之后，可对 SolrMeter 源码进行编译打包，如图 10-13 所示。

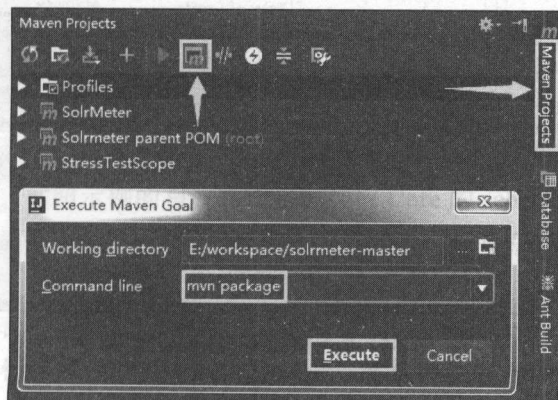


图 10-13 Maven 编译打包 SolrMeter 源码

如果你看到了如图 10-14 所示的提示信息“BUILD SUCCESS”，则表明 SolrMeter 源码



编译打包成功了。

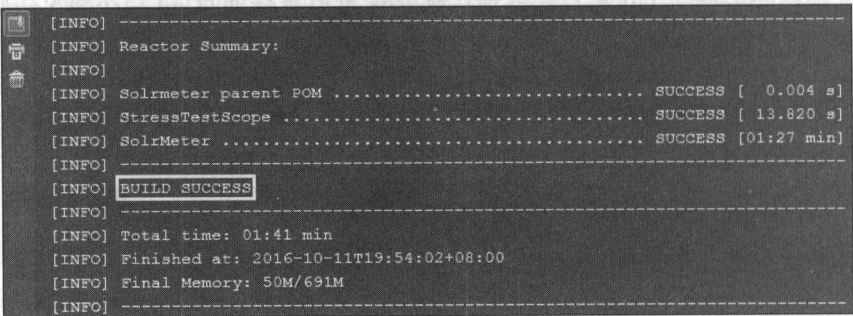


图 10-14 SolrMeter 源码编译打包成功提示信息

打包生成的可执行 jar 包存放在 solrmeter/target 目录下，文件名称为: solrmeter-{version}-jar-with-dependencies.jar。然后在该 jar 包的同级目录下创建一个 solrmeter.bat 文件，编辑文件内容如下所示：

```
java -jar solrmeter-0.3.1-SNAPSHOT-jar-with-dependencies.jar
```

然后你直接双击 solrmeter.bat 脚本文件即可运行 SolrMeter，运行后你将会看到如图 10-15 所示的用户界面。

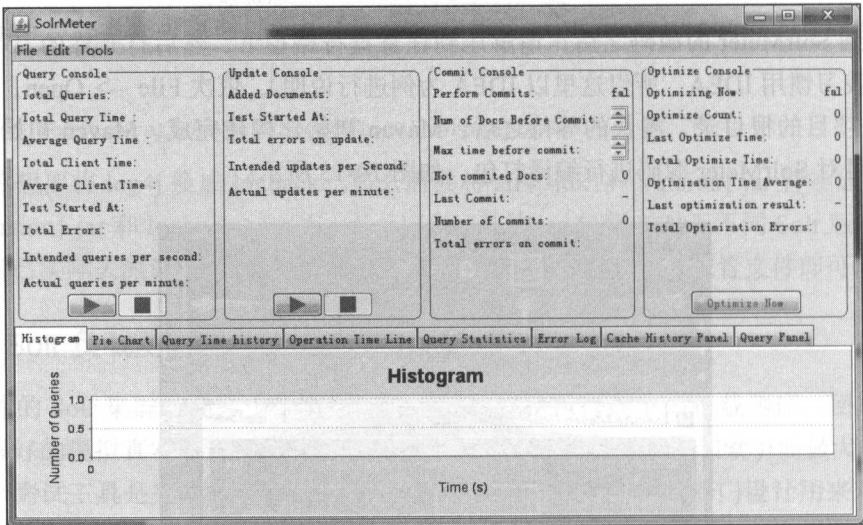


图 10-15 SolrMeter 运行默认界面

然后请单击菜单栏里的“Edit”，在展开的下拉菜单中选择“Settings”，然后你会看到 Solr 测试的配置界面，如图 10-16 所示。

Query Settings 是用于配置查询负载测试需要的一些参数，Update Settings 是用于配置



索引创建更新删除之类操作的负载测试。Optimize Settings 是用于索引优化配置，默认值是 ondemand 即按需优化。Statistics Settings 用于配置统计哪几项信息，比如下方的 8 个选项卡，依次表示历史各个时间点执行了多少次查询并以直方图的形式显示，查询耗时的饼状图，历史时间内的平均查询耗时直方图，统计每个操作的耗时（包括索引提交、更新、优化以及查询），每个查询具体详细信息统计，测试过程中的错误日志，缓存命中率折线图，查询面板（用于单个查询测试）。

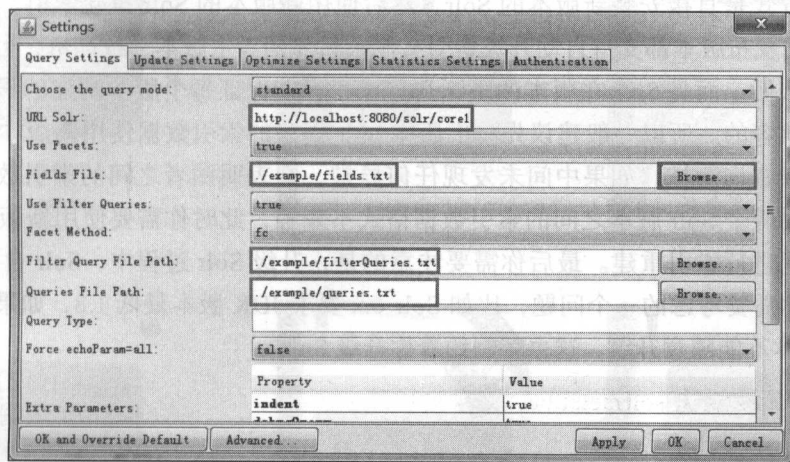


图 10-16 SolrMeter 负载测试设置界面

Query Settings 中比较重要的几个配置：URL Solr、Field File、Filter Query File Path 和 Queries File Path。URL Solr 参数很好理解，就是你要对哪个 Core 进行查询测试，那么你需要指定该 Core 的访问 URL；Field File 参数表示负载查询应该返回的域列表，使用外部文件的方式进行定义，其实就好比指定 Solr 查询中的 fl 参数；Filter Query File Path 参数表示用于配置你需要执行的 Filter Query 的查询表达式，即 SolrMeter 将使用你在文件中定义的 Filter Query 表达式来执行负载查询，同理还有 Queries File Path 就不言自明了。你可以单击右侧的“Browse”来选择你自定义的测试文件，这些测试文件该如何定义，其实你可以查看 SolrMeter 源码中 solrmeter\src\main\resources\example 目录下提供的几个示例文件，展开后看里面定义的示例，仿照示例的写法并根据你当前 Core 的 schema.xml 中定义的域稍作修改即可。一切配置完成之后，单击左下角的“OK and Override Default”按钮，否则不会生效。如果你配置正确了并且保存了，那么你会在主页面看到带有小三角图标按钮变成绿色的，如果是绿色的则表明你配置正确了，如果是红色的，则表明你配置有误。然后你就可以单击带有小三角的按钮开始执行查询负载测试了，主界面上的 Intended query per second 表示每秒钟执行几次查询，默认值为 1。

Update Settings 用于索引的创建更新删除的负载测试，比较重要的两个配置参数就是 URL Core 和 Updates File Path，具体如何配置，我就不多说了，因为 SolrMeter 源码中都有

提供相应的配置示例文件。

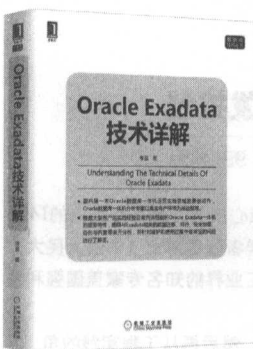
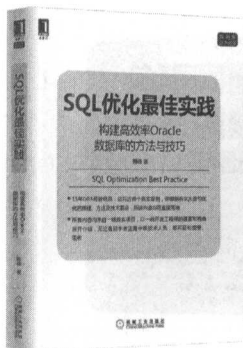
## 10.10 Solr 版本升级

Solr 的迅猛发展从它发版更新的速度就可以感觉得到,所以你可能需要时不时的升级你的 Solr 从而能够获取最新版本提供的新功能以及 Solr 自身版本更新后带来的性能提升。Solr 升级最好的方式是直接安装新版本的 Solr,然后使用新版本的 Solr 重建索引。然而 Solr 的每个官方正式发布版本都支持自动升级索引文件,只要两者主版本一致即可,但这并不意味着你必须这么做,而且 Solr 在版本的不断迭代中,不能保证每个版本构建的索引数据格式之间是互相兼容的。所以一般建议先将旧版本 Solr 创建的索引数据使用新版本 Solr 进行加载,然后进行查询测试。如果中间未发现任何异常,则表明两者之间的索引数据格式是兼容,否则表明两个 Solr 版本之间的索引数据格式不兼容,此时你需要使用新版本的 Solr 对所有索引数据进行索引重建。最后你需要注意的是,升级 Solr 过程中,Solr 自身对 JDK 版本的依赖是你需要考虑的一个问题,比如 Solr 6.x 要求 JDK 版本最低 1.8,如果你的项目依赖的 JDK 版本不能随意升级,这也就制约着你升级 Solr。

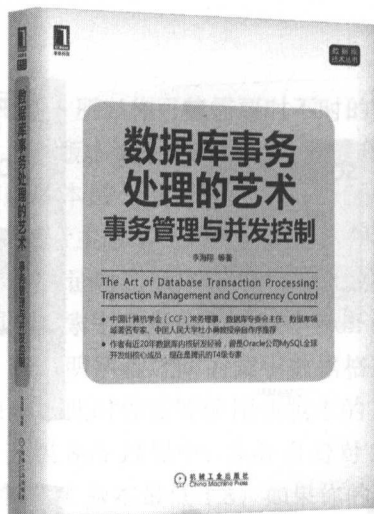
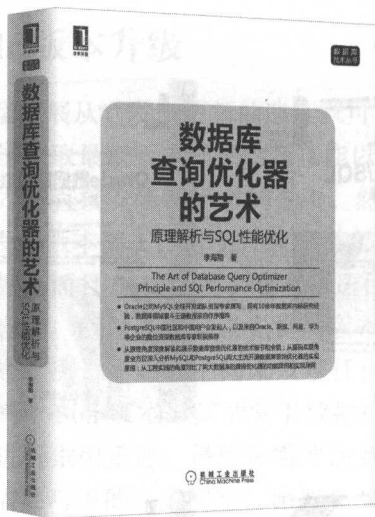
## 10.11 本章总结

在本章中,我们主要介绍了如何将 Solr 使用到正式的产品环境。其中讲解了如何编译打包属于你自己的 Solr 发布版本,并将 Solr 部署到企业生产环境中,以及部署过程中需要注意的一些问题。然后我们讲解了 Solr 的 Shard 和 Replcation 的个数如何确定,接下来我们讲解了企业中 Solr Core 应该如何管理,以及 Solr 集群应该如何管理,然后我们附带提及了如何实现与 Solr 进行交互,其中比较常用的方式就是 SolrJ,关于 SolrJ 的如何使用会单独一章进行详细说明。然后介绍了如何去监控你的 Solr 性能,可选的方式有 Solr Web 后台自带的界面、JMX、SolrMeter 等,最后我们稍微介绍了下如何对 Solr 进行版本升级。

## 推荐阅读



## 推荐阅读



### 数据库查询优化器的艺术：原理解析与SQL性能优化

作者：李海翔 ISBN: 978-7-111-44746-7 定价：89.00元

本书是数据库查询优化领域的里程碑之作，数据库领域泰斗王珊教授亲自作序推荐，PostgreSQL中国社区和中国用户会发起人以及来自Oracle、新浪、网易、华为等企业的数位资深数据库专家联袂推荐。

本书从原理角度深度解读和展示数据库查询优化器的技术细节和全貌；从源码实现角度全方位深入分析MySQL和PostgreSQL两大主流开源数据库查询优化器的实现原理；从工程实践的角度对比了两大数据库的查询优化器的功能异同和实现异同。它是所有数据开发工程师、内核工程师、DBA以及其他数据库相关工作人员值得反复研读的一本书。

### 数据库事务处理的艺术：事务管理与并发控制

作者：李海翔 ISBN: 978-7-111-58235-9 定价：99.00元

作者有近20年数据库内核研发经验，曾是Oracle公司MySQL全球开发组核心成员，现在是腾讯的T4级专家。数据库领域的泰斗杜小勇老师亲自为是本书作序，数据库学术界的知名学者张孝博士（中国人民大学）、卢卫博士后（中国人民大学）、彭煜玮博士（武汉大学），以及数据库工业界的知名专家盖国强和姜承尧等也给予了极高的评价。

全书共12章，首先介绍数据库事务管理与并发控制的基础理论和工作机制，然后再从工程实践的角度对比和分析了4个主流数据库的事务管理与并发控制的实现原理，最后通过源代码分析了PostgreSQL和MySQL在事务管理与并发控制上的技术架构。

## 作者简介

**兰小伟（网名：益达）** 资深Java工程师，在Java技术上有很深的积累和造诣。国内较早接触Solr的技术专家之一，长期致力于Solr的技术研究、实践和生产环境部署，是Solr社区的积极参与者和实践者，以让Solr技术能够在中国得到广泛应用不遗余力并乐此不疲。

现就职于国美金融，曾就职于各种大大小小的创业型公司。个人技术涉猎广泛，除了Java之外，对jQuery、ExtJS、AngularJS等前端技术也有研究。

技术宅，外表高冷安静，内心细腻感性，好文墨喜交友但不善交际。为人低调谦和，乐于助人，愿与各位志同道合者一同交流学习。



Solr是一个构建在Apache Lucene上的流行的、快速的、开源的企业搜索平台，它的主要功能包括强大的全文搜索、命中高亮、多维度查询与分析统计、丰富的文档解析、地理空间搜索、大量的REST API以及并行SQL。Solr是安全的、高度可伸缩的、可自动容错的分布式索引和搜索的企业级解决方案，并为世界上许多高流量的internet站点提供了全文搜索和导航的技术支持并备受欢迎。

Solr在企业内一个典型的应用场景就是电商商品搜索、类别导航区块、属性过滤区块、搜索框自动联想，等等。当下已是大数据时代，企业的业务数据量呈TB级增长，对于数据的搜索需求会愈加强烈，对于低成本的互联网企业，Solr的使用诉求也会更加普遍。

对于历史数据的查询，在数据量还不具规模的情况下，一般采用传统关系型数据库自带的索引功能即可实现高效的数据查询。但当数据上升到一定规模时，或许你会想到使用HBase数据库来救急，然而HBase目前只支持针对rowkey的一级索引，尚且不支持二级索引，此时Solr+Hbase的珠联璧合就可以完美打破这一局限性。但Solr的强大不仅如此，尤其当你足够了解Solr之后。

本书采用浅显易懂的语言加以适当的配图为你详细解读Solr的每个技术点，让其中涉及的每个原理、机制都不再晦涩难懂。理论结合实践才能出真知，案例驱动的方式贯穿本书始终，希望读者能够多上机实践书中的每个示例，遵循“理解为主，实践为辅”的学习原则，学以致用并在自己所在公司企业内部部署Solr，充分施展Solr的威力，从而体现自己的个人价值。

